4

Date of acceptance Grade

Instructor

Distributed Computing for the Internet of Things Using IoT Hubs

Janne Laukkanen

Helsinki November 17, 2017

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Janne Laukkanen

Työn nimi — Arbetets titel — Title

Distributed Computing for the Internet of Things Using IoT Hubs

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	November 17, 2017	59 pages + 4 appendices

Tiivistelmä — Referat — Abstract

The vast amount of data created in the world today requires an unprecedented amount of processing power to be turned into valuable information. Importantly, more and more of this data is created on the edges of the Internet, where small computers, capable of sensing and controlling their environments, are producing it. Traditionally these so-called Internet of Things (IoT) devices have been utilized as sources of data or as control devices, and their rising computing capabilities have not yet been harnessed for data processing. Also, the middleware systems that are created to manage these IoT resources have heterogeneous APIs, and thus cannot communicate with each other in a standardized way. To address these issues, the IoT Hub framework was created. It provides a RESTful API for standardized communication, and includes an execution engine for distributed task processing on the IoT resources. A thorough experimental evaluation shows that the IoT Hub platform can considerably lower the execution time of a task in a distributed IoT environment with resource constrained devices. When compared to theoretical benchmark values, the platform scales well and can effectively utilize dozens of IoT resources for parallel processing.

ACM Computing Classification System (CCS):
C.1.4 [Parallel Architectures],
C.2.4 [Distributed Systems],
C.4 [Performance of Systems],
J.7 [Computers in Other Systems],
C.3 [Special-Purpose and Application-Based Systems]

Avainsanat — Nyckelord — Keywords
Internet of Things, edge computing, fog computing, distributed computing

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Today, computers are ubiquitous: offices, factories, homes, public locations and transport vehicles are all filled with them. Also, people carry mobile computing devices like laptops, tablet computers and mobile phones everywhere they go, and are connected to the Internet the whole time. As a consequence, huge amount of data is created by these devices, which in turn could be used to offer smarter user applications. However, the data cannot readily be used by these applications without being preprocessed first, which requires a considerable amount of computing power. As novel artificial intelligence applications like Amazon's Alexa[1] and Apple's Siri[2] process voice commands in real time, and autonomous cars can generate 4000GB of data each day [Krz16], there is an even greater demand for the increase of available computing power. The devices, especially the mobile ones, do not have the resources to run computation intensive applications by themselves [HPR+13], so the data is traditionally sent to the cloud to be processed on a large amount of powerful computers.

In addition to the traditional computers, small, embedded computers have appeared into our surroundings even more thoroughly: watches, home appliances, network routers, furniture and even clothes can have sensors in them, and they are connected to the outside world via wireless links. This phenomenon is commonly called the Internet of Things (IoT), and it increases the amount of computers connected to the Internet dramatically. For example, numerous technology sources predict that there will be tens of billions of devices in the IoT in near future [Eri11, Eva11, Gar14].

Although the specific increase in the amount of data is impossible to foresee, moving all data processing to the cloud will be challenging, because the cloud platforms are often located far away from the devices at the network's edge, and thus introduce long latencies [AA16]. The problem is that there are applications built on top of the IoT that require low latency: humans are sensitive to delay and jitter in user-applications [SBCD09], and industrial IoT systems that manage process automation and manufacturing often have stringent latency requirements [ÅGB11]. Another challenge is that some data simply cannot be sent to remote servers for security reasons, e.g. when users want to keep their interactions with the local devices private, and not expose them to the cloud. For these reasons there is a need for the

[1] https://developer.amazon.com/alexa

[2] https://www.apple.com/ios/siri

data to be processed closer to the edges of the Internet, where the source devices are. Moreover, because devices in the IoT have constrained resources, any one device usually cannot process an offloaded task by itself. To make it viable to utilize the IoT devices for processing offloaded tasks, it has to be possible to process the tasks in a distributed and parallel fashion.

Systems for processing data in the edge networks already exist, and they are usually based on former research on cyber-foraging [Sat01]. For example, *cloudlets* [SBCD09] is an architecture where local computers could be utilized to process the tasks sent to them by nearby devices. The cloudlets are purposefully installed machines in the environment, and they use virtualization to enable execution of tasks that are offloaded from constrained devices. Another angle at the problem is taken by *Hyrax* [Mar09], which is a mobile computing platform that uses peer-to-peer communication to enable distributed task processing on mobile devices. However, these platforms have difficulties to be applied to the resource constrained devices of the IoT. Firstly, *cloudlets* are not designed to be a distributed processing platform, so processing on multiple devices in parallel would require further development. Secondly, traditional distributed computing platforms such as Hadoop[3], which is implemented in *Hyrax*, are designed for clusters of regular commodity computers, not for resource constrained devices, and thus energy usage is not as critical a factor for them as it is in the IoT.

While the traditional offloading systems are not optimal for IoT environments, current IoT middleware could be another candidate for distributed processing in the edge networks. Platforms such as ThingWorx[4], Xively[5], SensorCloud[6] and IoT Framework[7] have been created to handle the increasingly complex monitoring, management and control of the devices in a centralized way [MMST16]. However, the IoT devices in these platforms are usually seen only as producers of data, not as computational resources that could be utilized for external processing [HV15]. As a consequence, there are no standardized APIs for process offloading in the IoT, and neither the devices nor the IoT middleware are capable of exposing their processing power to third parties. This is unfortunate, because if the large amount of devices in the IoT could be utilized for distributed offloading, the data could be

[3]http://hadoop.apache.org

[4]https://www.thingworx.com/

[5]https://xively.com/

[6]http://www.sensorcloud.com

[7]http://www.ericsson.com/research-blog/internet-of-things/ computational-engine-internet-things

processed locally using the existing infrastructure, saving time, energy, and network bandwidth.

To provide a generic platform for exposing the data and compute resources in the IoT, the IoT Hub framework was created. The IoT Hub can be implemented to extend any existing IoT middleware or device to provide a standardized interface to IoT resources. It exposes a RESTful API for accessing and controlling the resources, which ensures interoperability between different IoT platforms and compatibility with modern web services. Importantly, the API also enables distributed task processing in the IoT resources. By using the IoT Hub in a network of IoT devices it is possible to create an environment of heterogeneous IoT resources that are all capable of communicating to each other, and processing each other's compute loads.

The challenge in creating a generic API to a wide range of different systems is to be able to adapt all the different ways of communication to a single API. In the IoT Hub this challenge is solved by introducing scripted software plugins called enablers, which connect the IoT resources to the IoT Hub API. Because they are implemented using a scripting language that is executed inside the IoT Hub, the enablers are independent of the platform implementation and can be re-utilized anywhere.

Another challenge in a distributed IoT system is mobility: the clients of the IoT resources could change locations frequently, and it is important to be able to update the information about available services. Service discovery in the IoT Hub is managed by the meta-hub [MT15] component. The IoT Hubs containing the meta-hub are regular IoT Hubs that can also act as registries for the services that the IoT resources provide, enabling advanced searching and service discovery capabilities. When an IoT Hub service provider wants to expose its services to third parties, he/she can publish the hub information to a meta-hub, which then makes it available to the IoT Hub clients.

The thesis contribution to the IoT Hub was to enable distributed processing of offloaded tasks in the platform. The IoT Hub achieves this by utilizing the script execution engine component, which is also responsible for executing the aforementioned enablers' code. External tasks that are to be processed in a hub are sent via the RESTful API, and they can also be distributed to multiple hubs simultaneously. When distributing the processing, each hub only processes its own share of the total data. The distribution process can also be nested, meaning that the tasks can be distributed again in the hubs that already received partial data, enabling further

sharing of load when necessary.

As a further demonstration of the thesis contribution to the IoT Hub, Appendix 2 includes a poster of my research presented at a workshop at the University of Helsinki, and also a paper is prepared to be submitted to the IEEE IoT journal[8].

To evaluate the IoT Hub platform's capability, it was implemented and tested on a distributed IoT testbed. Both distributed and non-distributed tasks were run on IoT devices, and performance benefits over processing in the original device were analyzed. Also, multiple levels of distribution were compared to a single level distribution, to measure the overhead of further offloading. The results show that distributed task offloading with the IoT Hub platform can considerably lower the execution time of a task in a resource constrained device. Also, the platform scales well when compared to theoretical benchmark values, and can effectively utilize dozens of IoT resources for parallel processing.

This thesis is structured as follows: Section 2 will introduce the state of the art of distributed computing, and the motivation for the IoT Hub. Section 3 shows the IoT Hub platform architecture, and details the thesis contribution. Section 4 presents the evaluation of the execution engine on a single desktop computer, as well as the IoT Hub's performance on a distributed IoT testbed. Finally, conclusions are presented in Section 5.

2 State of the Art of Distributed Computing

Distributed computing is a form of parallel computing, and distributed computing systems are used to solve problems that are too large to process in one machine, or to solve regular problems with smaller latency by dividing them to smaller pieces [C+05]. Distributed computing may also be defined as meaning computational resources that communicate through a network to coordinate their actions [CDK05]. To get a holistic understanding of distributed computing, it is good to know where it has come from, how it has developed, what is the state-of-the-art and what are the future prospects. To provide that, this section is structured as follows: Section 2.1 presents supercomputers, the early distributed systems. Section 2.2 describes cluster computing, which is the reigning form of supercomputing today. Section 2.3 presents grid computing and Section 2.4 defines the popular concept of cloud com-

[8] http://iot-journal.weebly.com/uploads/1/8/8/0/18809834/ieee_iot_si_rtdp_cfp.pdf

puting. Section 2.6.1 details the fog, a more recent form of computing utilizing local resources. The methods of distributing computations, or `offloading`, are presented in Section 2.5, and the current technologies that enable distributed computing on edge devices are inspected in Section 2.6.

2.1 Early Supercomputers

The need to run complex analytical tasks with computer systems has been driven by advanced research in areas like aerodynamics, weapons design and cryptography. What started as high-performance scientific computing later became called supercomputing, and it has been the most highly developed category of computing systems to use parallelism [Mac91]. The first customers to use supercomputers were national defense laboratories and scientific institutions, but later also large industrial corporations have started to invest in them [EM94].

Early supercomputers that implemented parallelism, such as the Cray XM-P[9], did so by using only a few processors, and utilized shared memory between the processors. These systems were eventually passed by massively parallel processors (MPP), which were the result of work by the Caltech Concurrent Computation Project [FOLR85]. MPPs typically use large amounts of processors, connected together with specialized networks to achieve low latency. They mostly dominated supercomputing until the end of the 1990s, and showed that supercomputing performance could be achieved with off-the-self microprocessors [Mat09].

2.2 Cluster Computing

Because the MPP type of supercomputers were expensive, and analyzing larger data sets in parallel fashion was becoming more common in the 1990s, there was a need for a more economic solution. Specifically, cluster computing was becoming a popular alternative for the MPPs at the start of the 2000s [BFG+00]. Cluster computing is a form of computing where, in contrast to MMPs, not only the processors are connected to each other, but complete computers instead. Also unlike the MPPs, which use specialized hardware for networking, regular networks such as Ethernet are used to connect the machines together. It's interesting to note that the idea of cluster computing was already developed in the 1960s, but it was the rise of

[9] en.wikipedia.org/wiki/Cray X-MP

computing power in regular PCs that made possible for commodity clusters to rival MPPs in the supercomputer market [Bak00].

Computers in a cluster, often called nodes, essentially work as one computer from the user's point of view, even though the processing happens in a distributed and/or parallel fashion [SK11]. Computers may be heterogeneous in terms of hardware and performance capabilities, but it's usually beneficial that they are as similar as possible because, in case of a failure, a task may be sent to any other node in the network. Clusters are managed through a central command point, which distributes the tasks to the worker nodes that perform the computations. Nodes are usually dedicated to work for the cluster only, and cannot be used for other tasks than what the cluster's managing node sends them.

An important aspect of cluster computing is fault tolerance [JPS13]. When many computers are connected together and they process a huge number of tasks, failures in the hardware are likely to occur, and they must be considered in any cluster computing system. Fault tolerance is usually achieved by replicating data to multiple nodes, so that when one node fails, another one can take its place and the task can continue to run on it instead.

Although cluster computing can lower the expenses of high performance computing, some situations require even more distributed form of computing. Specifically, certain scientific research applications may require that computing infrastructure from different geographical locations are used together, because each location may have specialized resources that cannot be found elsewhere. Another reason for the need to combine resources from different sites is that existing hardware can be utilized, without additional investment for building clusters.

2.3 Grid Computing

Grid computing was introduced in the mid 1990s [FZRL08], and was developed to enable utilization of distributed computing resources. Similar to clusters, grids aim to lower the costs of high performance computing by combining a large number of computers with high-speed networks. However, grids are more distributed in nature than clusters: while clusters usually host their computers in the same physical location to achieve low latency, the computers in a grid may be located in different geographical locations completely. Also unlike in clusters, different parts in the grid are usually heterogeneous, and work independently of each other.

Different parts of the grid are transparent, and the grid effectively looks like one computer to the end users. Grids share this transparency feature with the clusters, and some parts of the grids may also be implemented as clusters in the background. In contrast to clusters, the grid has no central command and works by aggregating distributed resources in an ad-hoc manner, and organizing them to work together to solve a common objective. After a task has been executed, the system that was setup for the task can disappear as fast as it was formed [SK11].

Groups of participants in the grid form entities that are called virtual organizations (VO). Because computational resources may reside under different administrative domains, security and interoperability are important. To achieve secure communication between participants, special middleware and standardized protocols have been developed and utilized [FZRL08]. While these protocols increase safety, they also mean that any organization or user desiring to benefit from the grid's resources will have to abide by the rules set for the grid, and is limited by them. Grids are thus not widely available resources in the Internet for anyone to use, but more like networks between partners, formed for mutual benefit.

Grid computing was initially developed to address the needs of scientific research, and it fits well for those purposes. However, one could argue that it has not been very successful in commercial use, possibly because it concentrates more on integrating different infrastructures and security than on being easily available as a service. Considering this it is interesting to note that the developers of grid computing have formerly stated that their target was to virtualize computing and information resources, to enable any entity to offer them for others to use [FK03]. So there was an intention to make grid computing available for everyone, but it could not answer the requirements comprehensively enough to become widely used. Out of the need to provide computing as a utility universally, another form of distributed computing emerged, driven by large Internet companies such as Amazon[10] and Microsoft[11].

2.4 Cloud Computing

In the 2000s, more and more companies and other organizations began offering their services in the Internet. One challenge in the Internet was the need for services' ability to scale quickly, because the demand for a service could increase very rapidly. Cloud computing came as an answer to this challenge by offering computing re-

[10] aws.amazon.com

[11] www.microsoft.com

sources as services for anybody, and offering pay-as-you-go pricing, where customers only had to pay for the time that the resources were actually used.

Cloud computing comprises the services that are offered as applications via the Internet, and the computational resources that run the service applications [AFG+10]. Cloud computing enables utility computing, which means that computing resources are readily available virtually at any scale, and can be utilized only for the time they are needed. Compared to the traditional computing where, to achieve scalability, large amounts of servers would have to be initially purchased to enable rapid scaling, cloud computing makes it possible to start with minimal computing resources, but scaling to massive amounts of servers automatically, if needed.

Cloud computing has five characteristics: on-demand self service, broad network access, resource pooling, quick scalability and seemingly infinite computing power [MG11]. On-demand self service means that cloud users can reserve resources from the cloud provider anytime they want, and automatically without human intervention. Broad network access means that the cloud provider's services are available through a wide range of client applications, like a web browser or a mobile phone application. Resource pooling refers to distributing the cloud provider's compute resources among the consumers, as required at any time of day. The resources are virtualized, and consumers usually can only specify the location of the physical resources on a higher granularity level. Rapid elasticity means that the virtual resources can be very flexibly scaled either up or down, depending on the current situation. For example, if the amount of concurrent users on a website increases rapidly, the cloud resources can be scaled up in minutes. Measured service means that cloud services monitor resource usage, and deliver real-time reports from them. This helps both the provider and the client to better plan their resource usage.

These five characteristics make the cloud a very appealing option also for distributed computing. Applications that are run in the cloud can be quickly distributed to any number of computers, because there are virtually endless resources. Latency also stays low, because the applications are running in the same environment as the processing servers. Many cloud providers are offering specialized services for distributed computing, such as Amazon AWS with Elastic Map Reduce[12] service, or Microsoft's Azure platform with its HDInsight[13] service. However, if applications are running in remote devices, such as mobile phones, network connections incur a

[12]https://aws.amazon.com/emr/

[13]https://azure.microsoft.com/en-us/services/hdinsight/

significant latency.

Despite the inherent latency with applications that run remotely and use cloud for some part of their processing, there are benefits for many scenarios where the latency is tolerable and there is a sufficiently good network connection. Mobile phones today have considerable computing resources, and more and more complex applications are run on them. However, complex processing requires more power, and battery technology has not been able to keep up with the new computational demands of mobile devices. Thus, utilizing cloud computing in mobile phones, or mobile cloud computing (MCC), has been developed to enable the execution of rich applications in mobile devices, and has been able to improve the service quality in mobile applications. MCC can be defined to mean the utilization of external resources in mobile devices for data storage and computational tasks [DLNW13]. In addition to remote clouds, MCC could utilize other external resources that are closer to the devices, but current applications are focused in utilizing the cloud [FLR13]. One reason for this could be that the computing resources closer to the mobile devices lack proper infrastructure and protocols for effectively handling externalized computing tasks.

2.5 Offloading

Offloading, the method of moving computing tasks from one computer to another, is an important part of distributed computing. Offloading is presented here to support the latter sections and to provide an overview of the different offloading models.

According to [FLR13], there are three main models for offloading: Client-Server Communication, Virtualization and Mobile Code, from which the two latter ones are the most prominent in current systems. Virtualization in offloading means that virtual copies of the original execution environment, including memory image, are deployed to remote machines without stopping execution. Virtualization has the benefit of being robust, because the virtual environment is effectively isolated, but setting up remote virtual machines incurs a relatively large overhead. Modern container-based virtualization could somewhat decrease this overhead, but in current IoT environments the devices have very limited resources, so utilizing virtualization is challenging.

Mobile Code approach, on the other hand, means sending only the actual application code and/or data to remote machines, and executing it in the remote machine's

environments. Mobile Code has been used to implement a cyber-foraging[14] system in a mobile cloud environment, which has proved that running applications on multiple machines in parallel is more efficient in terms of performance [Kri10].

Interestingly, there are some generic features in all offloading systems that have to be implemented, regardless of the offloading method. At a higher level, there are two concepts that can be offloaded: data and code. Furthermore, both data and code can be divided to smaller pieces for offloading, to utilize parallel processing. It is usually sensible to at least parallelize either data or code, to achieve performance increase in terms of latency. If energy usage would be the only concern, then only offloading without parallelization could be considered.

In scenarios where code is to be distributed, the initial device has to know what pieces of the code can be distributed, and what parts of the code must be executed locally. This, by far, is no easy task, because it usually requires either writing the application in such a way that parallelizable parts of the code are annotated, or static and dynamic code analyzers have to be utilized [FLR13]. Thus, for resource constrained environments, distributing the data seems like a better model to minimize the processing needed in the original device.

2.6 Fog and Edge Computing

Current challenges to computing are posed on one hand by the increasing number of mobile phones, and on the other hand by the rise of the IoT. Both of these phenomenons bring vast amounts of data and computing resources at the edges of the network. While mobile phones could, in some scenarios, have the necessary processing power and storage capabilities in the devices themselves, the resource constrained IoT devices do not [AIM10]. One solution to this is that the data is sent to the cloud for processing and storage, but also that has inherent problems. Because cloud computing has been addressing the problems that were generated by the need to scale traditional web applications, it is not suitable for the low-latency applications encountered in edge computing scenarios [CZ16].

The challenges faced in the IoT and in the mobile computing environments have created a need for a computing architecture that can bring enough computing power close to the edge devices, so that the latency requirements of modern, rich client

[14] Offloading workloads from resource constrained devices to stronger machines in local environment.

applications can be met [GLME+15]. Edge computing, meaning the utilization of near-by devices for offloaded data processing, can help to reduce the load currently imposed on the cloud. Also, an architecture called fog computing has been developed to address the issue. Because the edge devices are so central in fog computing, edge computing is used synonymously with the fog in this thesis, which is also usual in research [YLL15]. Furthermore, fog computing comprehends the IoT, although they are often mentioned separately here because of their distinct characteristics.

2.6.1 Fog Computing

In contrast to cloud computing, which refers to compute, storage and network resources that reside in a datacenter, fog computing provides an environment where those resources are distributed everywhere, essentially forming local mini-clouds from edge devices. The fog can address the latency and network connectivity issues present in the MCC, but is also capable of communicating with the cloud when needed.

Devices in the fog may range from mote-class devices that have very limited resources, to consumer smart phones and laptops, to a modern car with an abundance of sensors and numerous computers, to a small datacenter with significant compute power. Fog offers these resources for its users in the same manner than cloud, but with smaller latency. It also enables features such as using data from local sensors, which would not be possible in the cloud. However, it is often desirable to combine cloud and fog computing, because many applications also benefit from a global aspect in addition to the local one [BMZA12].

According to [VRM14], the emergence of fog computing can be seen as two-fold. Firstly, there are new technologies that have been developing independently of each other, and have now reached a mature level. Mainly this considers technologies around the IoT. Secondly, usage patterns of these new technologies suggest that some middle ground between having data in local constrained devices and cloud must be provided. One such pattern is the desire of users to better handle their privacy by not sending all their private data to a cloud service, where it might be exposed to unwanted third parties.

To give a broader view of the fog and how it differs from the cloud, a list of its characteristics is presented next. This list is a modification of a corresponding list in [BMZA12].

Locality Fog gives access to local assets, their data and compute power

Low latency Quick response to service requests

Heterogeneity Devices in fog may consist of very low-end devices, as well as enterprise class servers

Wireless access dominates Wireless networking is the main means of communication, possibly without Internet connection, using other wireless protocols locally

Geographic distribution Nodes in the fog can be dispersed on a large area

WSN support Using large Wireless Sensor Networks for monitoring data gathering, processing data locally

Real-time applications Deployment of applications that interact with the surrounding environment in real time

Because the fog is still a relatively new architecture, there are many challenges that have to be overcome to enable widespread adoption and to realize fog applications. Some important challenges are: node discovery, node management, security, standardization, monetizing and programmability [VRM14]. Node discovery and management imply that there has to be a way for nodes to register and make themselves discoverable among fog users. Also, the nodes need a way to be configured properly to function according to their tasks. Security in this regard means that when applications are running on multiple nodes, the environment needs to be secure, and users need to be sure that their data stays private. Security also requires that the hosts that run the applications must stay safe, usually implicating that a virtual environment is used for deploying applications. Standardization calls for a unified way to communicate between nodes, possibly by creating a standard API. Monetizing means that nodes should have some sort of motivation for offering their resources for third parties to use, for example giving the nodes free network access. Finally, programmability means that developing applications on top of the fog should be possible for developers without a big learning curve.

2.6.2 Current Challenges in Edge Computing

As mentioned in Section 1, numerous middleware systems have been created for the IoT. These systems are capable of monitoring and controlling sensors and actuators,

and offering managements services via web based interfaces. However, these systems have heterogeneous APIs, and are not designed for exposing the devices as compute resources. Thus, communication with them is not possible in a standardized way, and it is impossible to utilize their underlying computing power.

The fog architecture applied to the IoT can help to decrease latency and bandwidth usage by localizing data processing, avoiding the need to send data to the cloud and improving service quality [KBU15, ZCP+13, LGL+15]. However, utilizing local resources for processing in the fog has focused on more powerful machines, not on resource constrained IoT devices [YHQL15, ZCP+13]. This is unfortunate, because even though the majority of these devices are not comparable to even to a regular laptop, the sheer amount of them makes them a viable option. If computations are parallelized, the devices can be used collectively to form powerful distributed processing platforms. According to recent research [Kel16], the IoT devices can consume significantly less energy compared to an equally performant server or a laptop computer. Additionally, because the devices are already part of the local infrastructure, no further investments in purposeful devices are necessary.

The fog and the IoT thus both have challenges that must be addressed to enable resource efficient computing in the edge. A standardized API for communication to local devices would benefit both the IoT and the fog, and would enable unified application development on top of the edge devices. To be able to build such a generic API, the diverse group of IoT middleware needs another abstraction layer for accessing the IoT resources, and one that can also expose their computing power.

3 Distributed Computing on the IoT Platforms

The IoT Hub is an architecture for a generic, distributed IoT framework which provides a RESTful API for accessing the IoT resources' data and control features[MT15]. The IoT Hub provides a solution for the interoperability challenges in current IoT middleware by enabling the creation of a heterogeneous network of hubs, which are capable of communicating via a standard API, independently of the underlying hardware or software. The next section introduces the IoT Hub framework in detail, and is followed by the thesis contribution in Section 3.2.

3.1 The IoT Hub

Figure 1 presents the IoT Hub framework architecture. RESTful API is the external interface that is used to interact with the hub, and the mapping from URLs to IoT resources happens there. To abstract the resources, IoT Hub uses the concept of feeds. The feeds can be used to control what data, control and processing capabilities are exposed, how they are exposed, and to whom. Because the feeds are accessed through the RESTful API, access control to them can be implemented by using any secure method of authentication and authorization present in modern web applications. For example, access tokens like the JSON web token[15] could be utilized, and the hub's owner can decide what security measures are necessary.

The feeds also include metadata about the resources they expose, which can hold information about what type of data the feed has, who can access the data, dependencies to other feeds, when the data was updated and so on. The data types of the fields in the feeds can also have restrictions, so that incompatible types of data cannot be accidentally or intentionally combined, or otherwise misused. Basically the metadata enables a very fine-grained control of the underlying resources.

To enable the discovery of the hubs by third parties, the feeds' metadata can be published to special types of IoT Hubs called meta-hubs [MT15]. In addition to the feeds, the meta-hub functionality also resides inside the REST API component in Figure 1. Meta-hubs can be used to search and navigate the available feeds and services in the IoT Hubs, and to get the information needed to make API requests. They are a catalog of information in the same sense that Service Oriented Architecture (SOA) uses registries, and could be utilized using a standardized protocol such as the HyperCat [Lea13].

Importantly, the platform manages the IoT resources only via the feed abstraction, and does not directly interface with the devices or the middleware underneath. This is made possible by the use of enablers, which are software components that handle the integration of the IoT resources' interface to the IoT Hub. The enablers have two parts: first, the integration code that adapts the underlying device or system to the IoT Hub feeds (presented as `plugin` in Figure 1), and second, a custom configuration for the enabler in the specific hub (presented as `Plugin configuration` in Figure 1).

The code for the plugins is written with a scripting language, and is executed inside the script engine in the IoT Hub. The script engine is an important part of the IoT

[15] https://jwt.io/

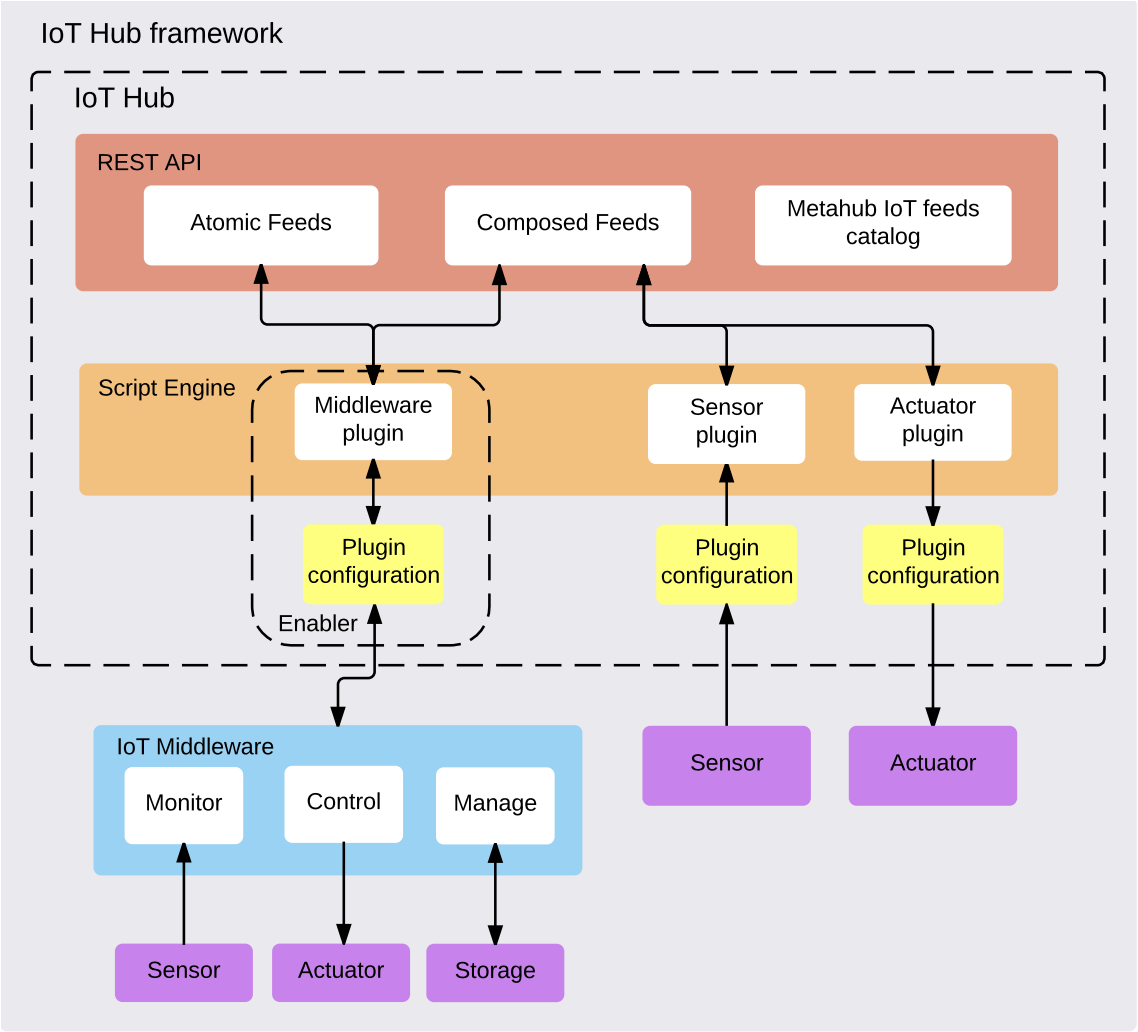


Figure 1: **The IoT Hub framework.** *Figure shows the logical components in the IoT Hub. Main direction of data flows is presented with arrows.*

Hub design and is shown as `Script Engine` in Figure 1. Because the plugins are executed inside the script engine, they are independent from the actual IoT Hub implementation language, and can be reused in any other IoT Hub implementation. Another important design factor related to the enablers is that they can be plugged in the IoT Hub dynamically during execution, so they need not be included in the hub beforehand.

3.1.1 Feeds

To provide a more in-depth understanding of the IoT Hub feeds, a description of their functionality is given here. There are two types of feeds that can be used

to access and control the IoT resources' data and control features in the IoT Hub: atomic feeds and composed feeds.

3.1.1.1 Atomic Feeds

Atomic feeds represent a single type of an IoT resource, which can typically be a sensor or an actuator. An example of a sensor type of an atomic feed is a simple temperature sensor, which returns a single type of value. When an atomic feed is used to control an actuator, the feed's value could simply be either `true` or `false`, representing a switch that can be turned on and off. Alternatively more varying values could be sent to control sophisticated devices.

Despite being a simple abstraction, atomic feeds include the metadata common for all types of feeds, and a powerful type system that can be used to add semantics to the data they expose. The type system enables strong typing of the IoT resources, and thus increases security and robustness of the applications developed on top of the platform. Also, atomic feeds act as an important building block for the more complicated forms of feeds.

3.1.1.2 Composed Feeds

Composed feeds are useful for two main purposes: for storing feeds' data persistently, or for combining and modifying other feeds. The storage capability could be used to gather real-time data from small sensor-devices using the atomic feeds, and then saving it for later analysis. The idea is to use the atomic feeds for real-time access to the sensors, and composed feeds for retrieving cached or historical values.

Combining multiple feeds into one is useful when a number of atomic feeds' data are related to each other and it is natural to process them together. Unlike the atomic feeds, composed feeds can contain different types of data: temperature values could be used together with humidity or luminosity. Thus, the composed feeds can provide a single endpoint for logically connected data, and also enable further modifications to the exposed data. Sometimes, for example, access to the raw values is not desirable. Using the composed feeds access to the sensitive data could be anonymised by only providing an approximate geographical location, like city, to hide the specific location of the data sources. Furthermore, the composed feeds can be used to combine data not only from atomic feeds, but also from other composed feeds.

Next, examples of how to use the REST API to create a composed feed is presented, as well as how to get and post data. Listing 1 shows how the feed is created by sending an HTTP POST message with the payload data specifying name, type and other information. The `fields`-parameter is used to describe the different types of data from which the composed feed consists of, and each object under the `fields`-parameter can have additional parameters that describe restrictions on that data type. An example of this is the `required` field, which specifies that the field is compulsory when using the feed.

Listing 1: Creating a feed

```
POST /feeds
----------------------
{
    "name": "temperature-feed",
    "type": "composed",
    "fields": {
        "temp": {
            "type": "Temperature",
            "required": true
        }
    },
    "timestamp": true
    "storage": "memory",
    "auth": "jwt"
}
```

Listing 2 shows how to send data to the feed that was created in Listing 1. The usage is simple, and only the field name and the value for the data have to be specified to insert data. After data has been inserted in the field, it can be retrieved by sending an HTTP GET message to the IoT Hub API, which is shown in Listing 3. The value also has an associated timestamp with it, because it was added as a requirement when the feed was created. Timestamps are useful for storing historical values, and the API could be queried for a specific value in time, values between two timestamps or all stored values, for example.

Listing 2: Posting data

```
POST /feeds/{id}
----------------------
```

```
{
  "temp": {
    "value": 24.6,
    "unit": "Celsius"
  },
  "time": {
    "value": 1496304294135,
    "unit": "ms"
  }
}
```

Listing 3: Getting data

```
GET /feeds/{id}
-----------------------
Response
[
  {
    "temp": {
      "value": 24.6,
      "unit": "Celsius"
    },
    "unit": {
      "value": 1496304294135,
      "unit": "ms"
    }
  }
]
```

3.1.1.3 Enabling Computing Resources in the IoT Hubs

Importantly, despite the generic nature of the atomic and composed feeds, they lack the capability for the users to use custom methods to access the feeds. This means that the feeds cannot be used by third parties to flexibly change the output of a feed, but would be restricted by how the feed was originally configured. While this is good in many situations, as it increases the robustness and security of the applications, it also leaves no room for easy customization of the interface, which

is important when applications are developed on top of the feeds. Moreover, the atomic and composed feeds cannot be used to execute offloaded code from external sources, and thus can not contribute to the problem of bringing computing resources nearer the IoT resources. For this reason, another type of feed was developed to extend the IoT Hub's capabilities to execute custom code inside the hubs.

3.2 Thesis Contribution: Executable Feeds and Execution Engine

As previously noted, the amount of devices in the edge networks is rising rapidly, and calls for new ways to process the data closer to the source devices. The thesis contribution to the Iot Hub consists of developing the executable feeds, as well as extending the script engine to execute code scripts from external requests. They are an important addition to the IoT Hub framework, and enable user-customized feeds, offloading of tasks and distributed computing in the edge devices. The executable feeds' main role is to expose the computing resources of the IoT devices through a RESTful API, similarly to how atomic and composed feeds expose data and actuator resources.

Computing resources are utilized by sending code scripts to the API, which are then executed in the execution engine inside the IoT Hub. The scripts can be fully defined by third parties, and thus enable extreme customization possibilities. The executable feeds also enable distributed processing by dividing the data used by the scripts to pieces, and then distributing the data and the code to remote IoT Hubs. The already distributed pieces of the data may also be distributed further, making it possible to only use a limited amount of hubs for initial, first level distribution.

Figure 2 presents the IoT Hub architecture with the thesis contribution included. The figure is similar to the Figure 1, with the exception that an `Executable Feeds` component is added to the REST API, and a `3rd party code execution` component is included in the script engine. Also, interactions between other feeds types and plugins have been omitted, to clarify executable feeds' interactions.

The executable feeds component handles the API calls, and prepares the environment for the script execution. The preparation can include retrieving of data beforehand, and storing it in memory so that the executed script has access to it. If the data sources are not known beforehand, they can be queried from the meta-hub catalog, which may or may not reside in the same hub. If the data needs to be

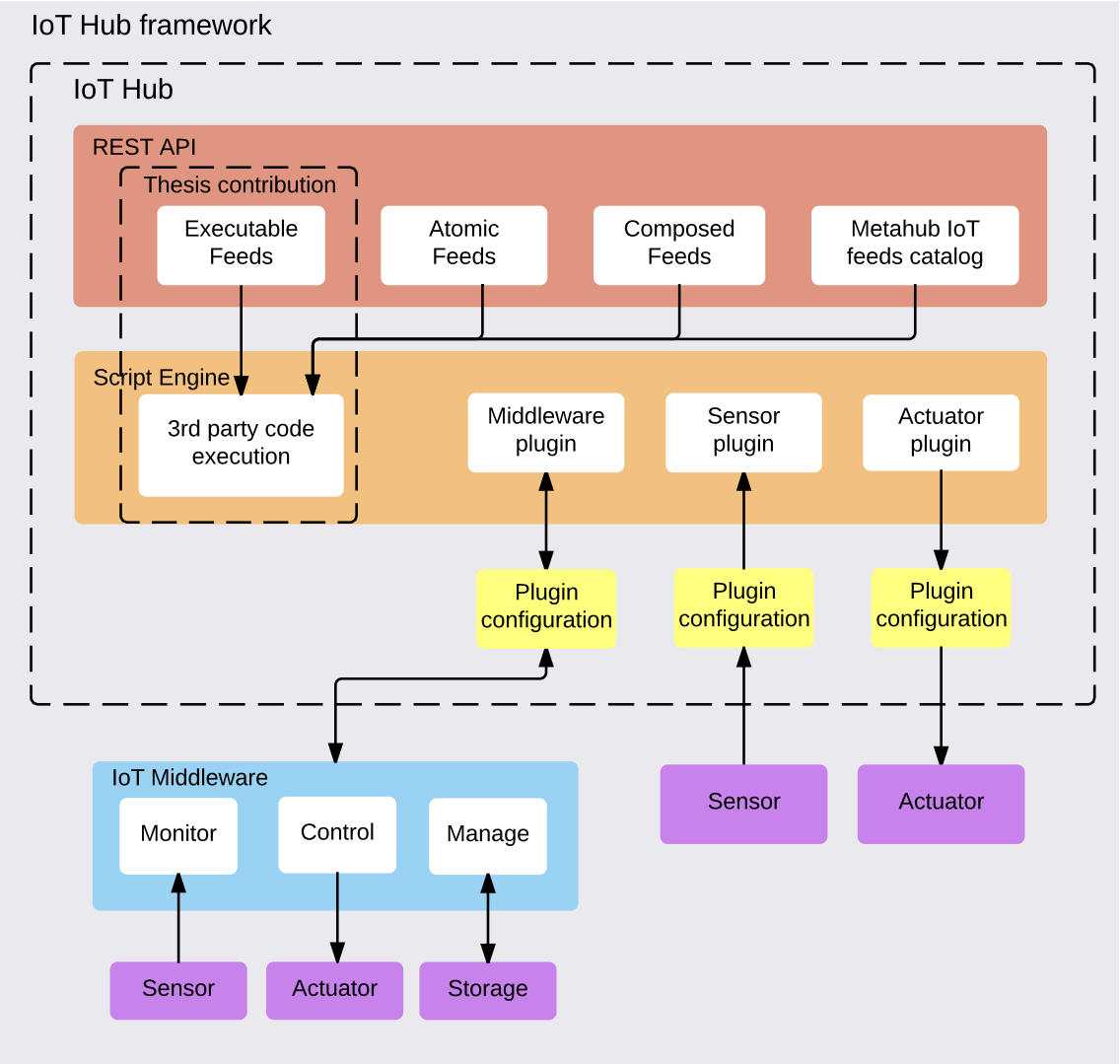


Figure 2: **The IoT Hub with executable feeds** *Figure shows the IoT Hub components with the thesis constribution included.*

distributed, the distribution also happens in the executable feeds component, before the script is passed to the execution engine.

The IoT Hub executes the scripted code inside a sandboxed environment, which is isolated from the hub itself to protect it. The sandbox also gives other benefits than security, such as modifying the compute resources that are available to individual processes. Depending on the situation, the engine could be used to utilize the underlying compute resources fully, or to only use some fraction of them. Because the hubs could reside in an environment where there are multiple clients utilizing the same hubs' resources, a fully dedicated allocation of processing power to a single client could be unfeasible. In such situations the hub could limit the amount of

CPU, memory and other resources available, to provide a fair distribution of the resources.

To control how the data is processed, executable feeds feature metadata that can be described when the feed is created, or with each individual API request. The idea is that if the metadata is given during the feed creation, the feed owner can restrict the feed behavior to some wanted subset. If a more flexible usage of the feed is desirable, only a minimal amount of information can be described for creation, and the rest can be left for third parties to define with each API request. Metadata for the executable feeds includes the following information:

1. Name/identifier of the feed
2. Authentication information
3. Data sources made available to the code
4. The payload code
5. Functions for distributing/combining data

The first two items in the list are common metadata that exist for the other feed types as well. However, the last three are specific to the executable feeds. First, the script code that will be executed inside the hub can be given. A common use case could be to calculate new results from a given input, to create custom results. Importantly, the scripts can use data from other feeds as inputs. Second, data sources that are available to the payload code can be defined. The data sources are defined as other feeds, which may be atomic, composed or other executable feeds. If no data sources are defined, the payload code cannot utilize other feeds' data in the execution. Lastly, two special types of functions can be defined, which are related to the distribution of the tasks: `mapping` and `reducing` functions.

In the IoT Hub and in this thesis, the concept of `mapping` is defined as meaning the process of dividing data to pieces to be distributed to remote hubs, and the concept of `reducing` to mean the process of combining the returned results from the remote hubs. Although these names are similar to the ones used in the Google's MapReduce [DG04] algorithm, the IoT Hub's executable feeds do not implement the MapReduce algorithm, and the mappers and reducers can be used in any way desired to distribute and combine the data. Specifically, there are no restrictions on how the functions must read input or provide output of the data.

Mapping of data in the executable feeds can be done in two ways: by data mapping or by URL mapping. Data mapping means that data is distributed by first fetching the whole data set, and then dividing the actual data to smaller pieces to be sent to the remote hubs. This kind of distribution is quite straightforward, and relatively easy to implement. However, data mapping also incurs a large amount data to be sent over the network, which can become a serious bottleneck for larger amounts of data.

To avoid the pitfalls of data mapping, a URL mapping scheme was devised for the IoT Hub. URL mapping means that the whole data does not need to be fetched by the original, distributing hub, but that the data sources can be defined as URLs instead, which refer to partial data fetched from other remote IoT Hub feeds. This implies that instead of initially fetching a large amount of data and sending the pieces to remote hubs at once, the pieces can be fetched by each remote processing hub independently. URL mapping thus avoids sending large amounts of data initially, and instead distributes also the initial data acquiring part of the distributed process.

Finally, a description of two different processing flows is presented for clarification in Figure 3. First, a processing task which is not distributed to other hubs and second, a task that is distributed to external hubs.

4 Evaluation

To evaluate the performance of the IoT Hub, it is primordial to test i) the performances of the local execution engines (referred to as EE for the remainder of the thesis) and ii) the impact of the network topologies and the mapping schemes on the distributed computation. The EE tests were run first, and after their results were analyzed, the implementation for the distributed tests was chosen.

Four different metrics were used to benchmark the performance: latency, CPU usage, memory usage and payload size. Latency is an important indicator of performance, because it demonstrates how fast the distributed processing is compared to local processing, and how large overhead the distributing part of the application incurs. CPU usage is important mainly for energy usage estimation, so that the tests can give some clues as to how large energy savings are possible by distributing processing. Memory usage is important for resource constrained devices, which IoT environments often consist of, so that fitting execution engine for the payload script code could be chosen. Payload size can help to decide how the data is distributed

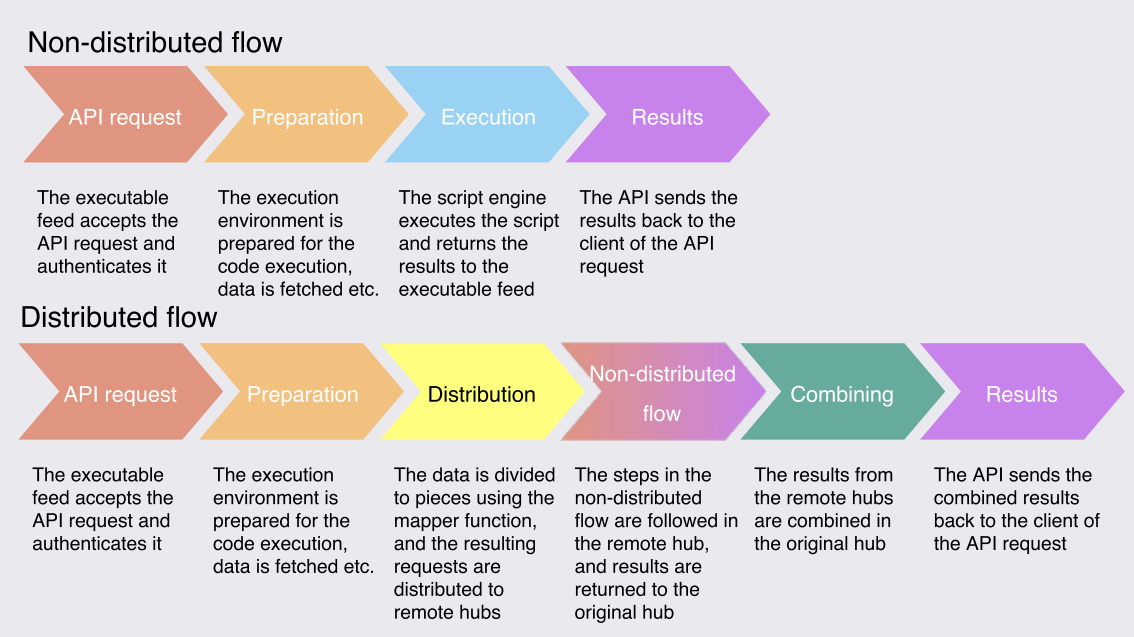


Figure 3: **Non-distributed and distributed processing steps** *Non-distributed flow shows the steps when data is not offloaded to other IoT Hubs. Distributed flow is more complex, and implicitly also includes the same steps that the non-distributed flow has.*

and what are the network bandwidth requirements.

In addition to the afore mentioned metrics, profiling information was collected during execution and it was used to determine what parts of the program execution were becoming bottlenecks when more hubs were added. Profiling information was also useful when calculating the theoretical maximum for the parallelizable portion of execution. Lastly, Amdahls's law, presented in Section 4.1, was used as a benchmark when the IoT Hub results were compared to theoretical parallel processing limits.

Here is a description of the four metrics:

Latency Latency of each request / response pair sent to a remote Hub

CPU usage Mean CPU usage in original Hub during task execution

Memory usage Mean memory usage in original Hub during task execution

Payload size Mean payload size of requests and responses sent between the Hubs

Next the setup and results for both test sets is presented. Performance evaluation

for local execution engine tests is shown in 4.2, the distributed test set is presented in 4.3 and lastly, a summary of the results is shown in 4.4.

4.1 Analysis Methods for Parallel Processing Scalability

In the past, analysis methods for parallel processing have been created to enable predicting application performance in distributed environments. Two most well-known frameworks for scalability analysis are Amdahl's law [Amd67] and Gustafson's law [Gus88]. Amdahl's law describes how parallel computing can theoretically scale with respect to how large part of the program execution can be parallelized. The law assumes that some constant portion of the computation is always sequential, and thus the total execution cannot be made faster no matter how many processing units are used for the parallel part. For example, if 95% of a program can be parallelized, the program cannot become more than 20 times faster, because 5% of it has to be processed sequentially.

Gustafson's law extends Amdahl's law by stating that when the processing capabilities of a system increase, the problems that are solved with the system also tend to get bigger. This means that the limit in latency exposed by Amdahl's law for a certain application can be exceeded by increasing the relative size of the parallel portion. As an example, if the aforementioned 95% parallelizable program uses a dataset sized 1000 elements for the parallel execution, the speedup factor limit of 20 provided by the Amdahl's law may be exceeded by increasing the parallelizable dataset size to 100000 elements, because it decreases the relative portion of the sequential part of the program.

Both laws are tools for analyzing parallel processing, and it depends on the use case which one suits the scenario better. In practice there is always an overhead incurred when computations are offloaded, so actual processing speed cannot easily achieve theoretical limits. However, the theoretical limits present a good benchmark against which different offloading and distributing methods can be compared. The closer the real performance results are to the theoretical ones, the better the system scales.

Because Amdahl's and Gustafson's laws were initially applied to multiprocessor environments where slower networks were not present, the laws' applicability to modern cloud environments have been researched more recently. For example, [dRSGS16] examines Amdahl's law's application in cloud environments, and [STP+12] uses both laws for predicting High Performance Computing (HPC) Cloud performance. Inter-

estingly, the latter research suggests that to be able to create accurate predictions, the communication part of the processing has to be handled separately.

4.2 Performance Evaluation of Execution Engines for the IoT Hub

The EE tests were implemented to measure the performance of different JavaScript execution engines. This was relevant because the script engine executes the payload code, so its performance affects the total performance of the IoT Hub considerably. The tests also aimed to show how much the implementation language of the IoT Hub itself affects the total performance.

Before running the EE tests, there were two implementations of the IoT Hub: a Java based implementation, called the Kahvihub [MMST16], and a NodeJS one called the Solmuhub. Both implementations use a JavaScript engine to run the payload code. JavaScript was chosen because of its widespread usage in the Internet, and because most IoT middleware systems offer a web based interface that can readily use JavaScript.

The two IoT Hub implementations use different JavaScript engines for code execution. Kahvihub uses the Duktape[16] engine, which is an embeddable JavaScript engine that focuses on portability and small memory footprint, and Solmuhub uses NodeJS's built-in V8 engine[17]. Because Kahvihub is based on Java and thus does not have an embedded JavaScript engine like in NodeJS, it has to use the Java Native Interface[18] (JNI) to enable JavaScript execution For this reason it is not simple to add support for other JavaScript engines to the Kahvihub implementation. This was one factor that later affected the decision of which implementation to use for the distributed tests.

To test the different characteristics of the hubs, three different algorithms were used in the EE tests: Quicksort, Fibonacci and Newton's method. The Quicksort algorithm is used to measure how efficiently the implementations handle memory access by sorting arrays of floating point numbers. The Fibonacci algorithm measures performance in handling nested function calls, and Newton's Square Roots method calculates the square root of a number, and is used to assess how well the IoT Hub

[16] http://duktape.org/
[17] https://developers.google.com/v8/
[18] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/

implementations handle large amount of arithmetic operations.

4.2.1 Experimental setup

The EE tests were implemented on a Fujitsu Esprimo E5731 desktop computer using an Intel Core 2 Duo E7500 processor, with 4GB of RAM and no remote connections. The dual core processor was used in single core mode to enable more accurate CPU measurements. Operating system used was Ubuntu 14.04 LTS.

Four different environments were used to run the algorithms:

Java-Duktape Java implementation of the IoT Hub, called Kahvihub, with Duktape JavaScript engine

NodeJS-Solmuhub NodeJS version of the IoT Hub, called Solmuhub, with native V8 JavaScript engine

NodeJS-Duktape A NodeJS script that uses Duktape engine to evaluate JavaScript code

NodeJS-V8 A NodeJS script that uses the V8 engine to directly evaluate the payload JavaScript code

The last two items in the above list do not implement the IoT Hub's functionality. More precisely, they are just small NodeJS applications that can receive HTTP requests, and take the payload code and execute it either directly in the V8 engine (NodeJS-V8), or inside a NodeJS virtual machine using the Duktape engine (NodeJS-Duktape). They are used to measure how large overhead the IoT Hub implementation itself adds regardless of the used JavaScript engine. For example, if the Java-Duktape would be slow, but NodeJS-Duktape would be fast, then it would indicate that the Java implementation is less performant than the NodeJS one.

Metrics used for the EE tests were latency, CPU usage and memory usage. Of the four metrics presented earlier, payload was not used because the tests were run locally on one machine, and no data or code needed to be offloaded. Latency was measured by sending 200 similar requests to the hubs and calculating the average response time. CPU and memory usage were calculated by using an external script that recorded CPU and memory usage in constant intervals during the execution.

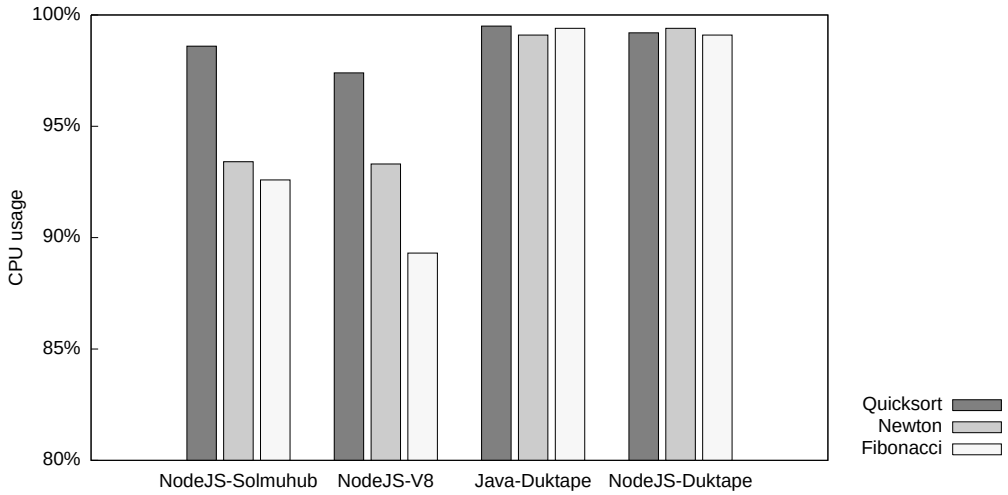


Figure 4: **Mean CPU usage in all hubs.** *Mean CPU usage is high in all IoT Hub implementations. However, NodeJS-V8 and NodeJS-Solmuhub use 5-10% less CPU on average than Java-Duktape or NodeJS-Duktape versions*

4.2.2 Results

Figure 4 shows the mean CPU usage in all hub types during execution. CPU usage in all four test implementations is high (over 90%), because the algorithms are run repeatedly and all of them are CPU intensive. However, there are some differences between the implementations that use the Duktape engine and the ones that use the V8. Both applications that use the Duktape engine have a very high CPU usage for all algorithms, between 99,1% and 99,5%. The implementations that use the V8 engine have a lower mean CPU utilization, ranging from 89,3% for NodeJS-V8 processing the Fibonacci algorithm, to 98,6% for NodeJS-Solmuhub running the Quicksort algorithm. The difference is especially clear for the Newton's method and Fibonacci algorithms.

These results show that the V8 engine has an advantage over the Duktape engine when processing power is important. More processing power means more computations can be completed in a less amount of time, so less energy is consumed. When there are large amounts of IoT Hubs running, even a 10% decrease in CPU usage may result in significant savings, so the results are relevant also in that sense. Furthermore, because the Duktape engine is less performant than the V8, it also takes longer to execute a request, which means that more CPU power is used for a longer time, resulting in even larger total energy consumption.

Figure 5 shows memory consumption in the original hub during distributed execu-

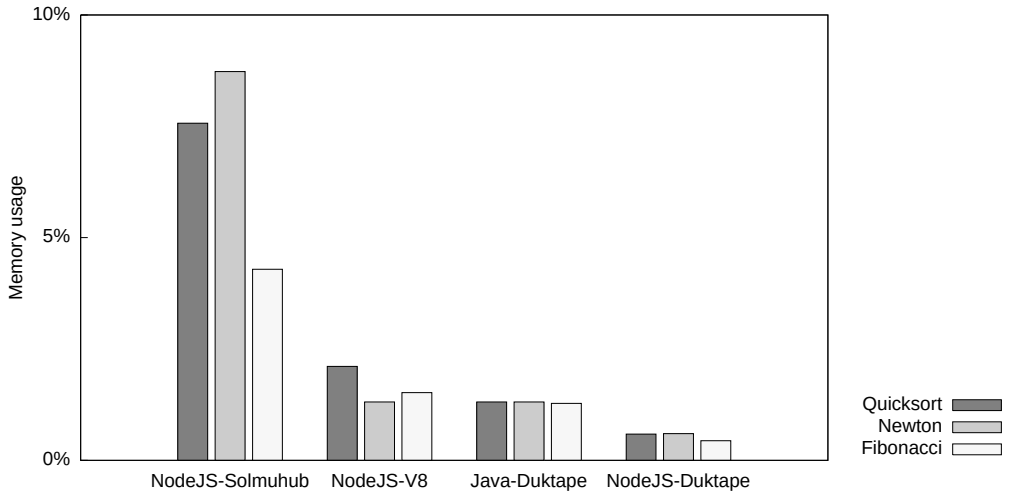


Figure 5: **Mean memory usage in all hubs.** *Mean memory usage is considerably higher in NodeJS-Solmuhub than in other hubs. Versions that use the Duktape engine have the lowest memory footprints. Because NodeJS-V8 does not have to load IoT Hub application modules like the Solmuhub, its memory footprint is also smaller.*

tion. Contrary to the CPU usage tests, memory usage is relatively low on each IoT Hub implementation. Importantly, the applications using the Duktape engine are clearly more efficient in memory utilization.

`NodeJS-Duktape` is the most memory efficient, using only from 0,4% to 0,6% of memory for all tests. Next is `Java-Duktape`, which uses from 1,2% to 1,3%, and `NodeJS-V8` with a range from 1,3% to 2,1%. `NodeJS-Solmuhub` is the last one, utilizing from 4,3% for the Fibonacci to 8,7% for the Newton's method.

The memory results show that when the Solmuhub functionality is implemented, there is an increase in the memory consumption, which is expected. However, it is interesting to note that the `Java-Duktape` implementation is more memory efficient than the plain `NodeJS-V8` execution of the payload code. So even with the overhead of the Solmuhub application, the Java implementation uses less memory than `NodeJS-V8`. This might be relevant for systems that are resource constrained in terms of memory, and can tolerate higher latency.

Latency results are divided into three different graphs. Each graph shows the performance of all four applications using one of the three algorithms. `Java-Duktape` implementation is used as the benchmark, against which the other implementations are compared to, so it's value is always 1. The other implementations are then

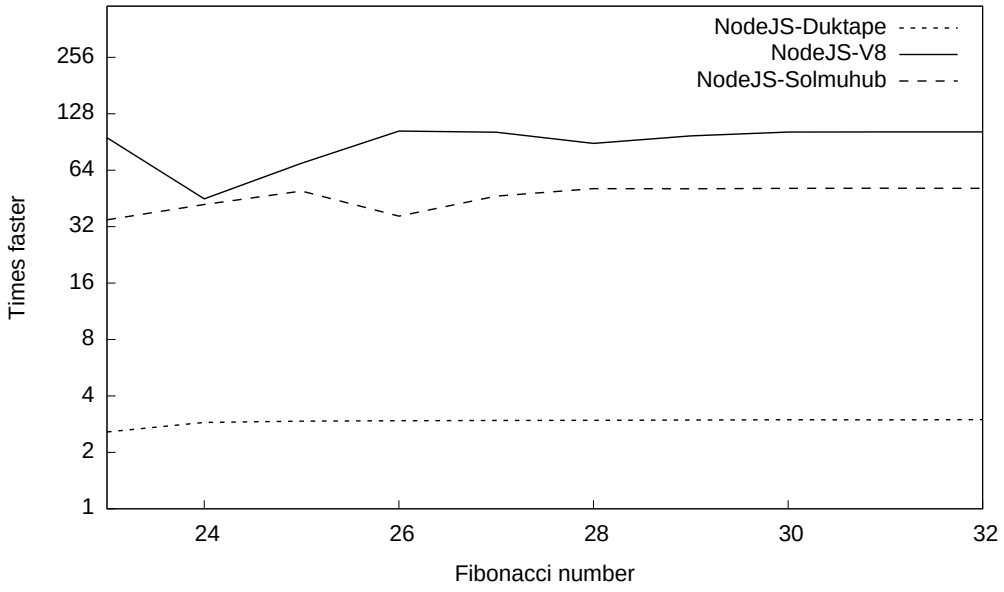


Figure 6: **Nested function calls with the Fibonacci algorithm.** *Both hubs that use the Duktape engine are considerably slower. NodeJS-V8 is about 100 times faster than Java-Duktape, and NodeJS-Solmuhub is over 30 times faster.*

mapped to the graph according to how fast they are in comparison to the Java version. Logarithmic scale is used to present each hub's results more clearly.

Figure 6 presents the results for the Fibonacci algorithm. The algorithm creates many nested function calls, which can be important for applications that need to perform recursive function calls.

`NodeJS-V8` application is the fastest one, being a little over 100 times faster than `Java-Duktape` when calculating Fibonacci numbers greater than 25. `NodeJS-Solmuhub` is the next fastest, with results varying from 36 to 51 times faster than `Java-Duktape`. Lastly, `NodeJS-Duktape` is constantly 2,8-2,9 times faster than the benchmark application.

These results imply that the V8 engine's performance for nested function calls is considerably better in terms of speed than the Duktape engine. Thus, for more complicated applications that include nested function calls and require more processing power, the V8 engine is a better choice.

Figure 7 shows the Quicksort results, and they are very similar to the Fibonacci algorithm's results in Figure 6. Quicksort sorts arrays of different sizes with the quicksort sorting algorithm, and is a good benchmark for memory access speed [KLD12]. Distributed sorting of a large dataset could be a relevant application in the real world,

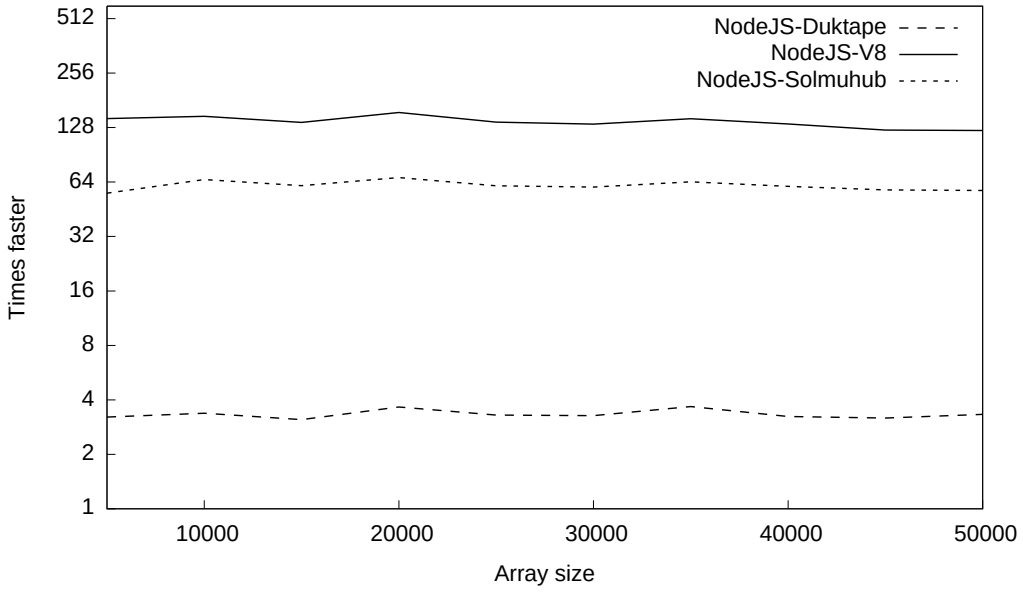


Figure 7: **Memory access speed with the Quicksort algorithm.** *Hubs using the Duktape engine are much slower when assessing memory access speed with quicksort.*

so these results are important.

The order from the fastest to the slowest stays the same: `NodeJS-V8` is the fastest, then `NodeJS-Solmuhub`, followed by `NodeJS-Duktape`, and `Java-Duktape` is the slowest. This time, `NodeJS-V8` is 123-148 times faster and `NodeJS-Solmuhub` 57-66 times faster than `Java-Duktape`. `NodeJS-Duktape` is again a little faster than the benchmark, with 3,2 times faster results than `Java-Duktape`.

The results imply that the V8 engine is superior to the Duktape engine when the application needs quick access to memory structures, and that NodeJS has an advantage over Java in JavaScript execution also in this sense.

The last one of the EE tests is the Newton's method algorithm, which iteratively calculates the square root of floating point numbers. This test measures how good the implementations are handling arithmetic calculations. In an IoT environment an application that needs to perform heavy calculations based on sensor readings, for example, would be a good example of an application that matches the Newton's method test.

Figure 8 shows that the results are again similar to the two former tests. However, `NodeJS-V8` and `NodeJS-Solmuhub` are closer in performance to each other in this test, ranging from 86-105 times faster results for `NodeJS-V8` and 64-92 times

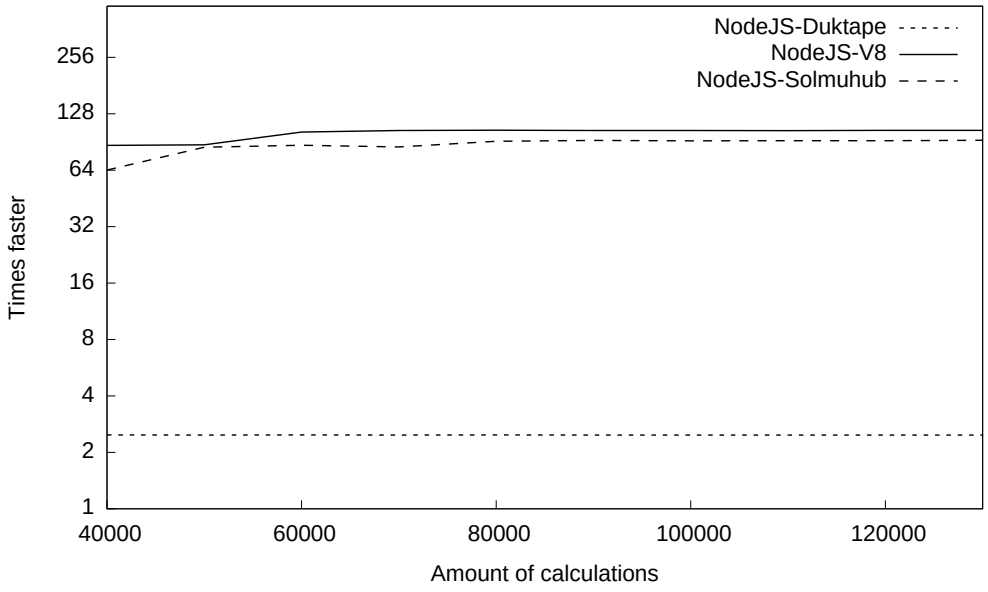


Figure 8: **Arithmetic calculations with the Newton's method algorithm.** *NodeJS-V8 execution and Solmuhub are much faster than versions using Duktape. Both NodeJS versions are almost 100 times faster than Java with Duktape.*

faster for `NodeJS-Solmuhub`. `NodeJS-Duktape` is constantly 2,5 times faster than `Java-Duktape`. This test continues to show that the V8 engine is well above the Duktape engine also when performing raw arithmetic calculations.

In conclusion of the EE tests, it is clear that the V8 engine would be a better choice for the distributed processing tests because of its performance advantage. In the future it would be interesting to compare other JavaScript engines that would be faster than Duktape, but also more memory efficient than the V8. However, the choice was made to implement the distributed processing tests with `NodeJS-Solmuhub`, using the V8 engine also as the execution engine for the payload code. In addition to being significantly faster than `Java-Duktape`, `NodeJS-Solmuhub` also has an advantage when multiple different JavaScript engines are tested, because they can be integrated in NodeJS as modules in the main IoT Hub application, and thus for the payload execution with little extra work. As mentioned, the Java implementation has to use the JNI to integrate other JavaScript engines, which requires considerably more work to implement.

4.3 Performance evaluation of IoT Hubs for distributed computing

Distributed processing tests were created to measure how well the IoT Hub performs when the processing is offloaded to remote machines using HTTP requests, and how well the implementation scales when multiple IoT Hubs are used and larger amounts of data are sent over the network. For these reasons, interesting research questions for this distributed IoT environment are:

Q1 How to decide when offloading makes sense

Q2 How to make data available for remote hubs

Q3 What are the strengths and weaknesses of various distribution topologies

To be able to answer these questions, the tests were designed and implemented in a way that the distributed processing could be compared to local processing in a single machine, and so that the different models of making data available to the hubs could be assessed. Also, one-hop distribution, meaning that the processing is distributed only one-hop away from the initial IoT Hub application, was compared to multi-hop distribution, where the processing applications may further distribute the processing to other hubs.

Distribution methods

To distribute the data to the IoT Hub applications (hubs from now on), the two mapping methods described in Section 3.2, URL mapping and data mapping, were utilized. In data mapping the whole test data is fetched in the original processing hub, then split into as many pieces as there are remote hubs, and sent to each hub along with the payload code. When using URL mapping, a URL which describes where the data can be fetched is sent in the request, instead of the actual data.

To demonstrate a scenario where a certain part of an image is fetched, the following example shows the URL's structure: `/feeds/1?size=1024&nodes=4&index=2`. This URL specifies that an image of size 1024x1024 pixels is requested (size=1024), the image needs to be split into four pieces (nodes=4), and that the piece number two is returned to the caller (index=2).

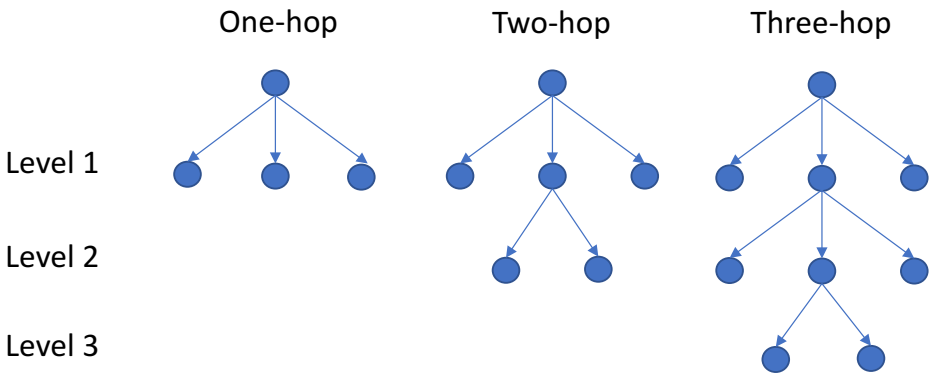


Figure 9: **Distribution topologies** *The three different distribution topologies: one-hop, two-hop and three hop present distribution of processing respectively to one, two and three levels.*

Distribution topologies

The two different distribution topologies, one-hop and multi-hop distribution, are shown in Figure 9. The levels used for multi-hop distribution were two and three, because those levels were adequate for getting information about how the system would scale for nested scenarios, but were also simple enough to implement with the limited amount of devices.

Multi-hop distribution was important to test because of its capability to overcome the challenges in scenarios when there are not enough processing hubs available one-hop away, or in scenarios where only a small amount of hubs is available for initial distribution, but the first-level hubs may use their own connections to other hubs for further distribution.

4.3.1 Testbed setup

The testbed consisted of 15 Raspberry Pi 3s and 2 Raspberry Pi 2B+:s, connected to a 24-port HP Procurve 2524 switch with Ethernet cables. Network bandwidth was effectively limited to 100MB/s, because the Raspberry Pis do not have a Gigabit Ethernet connection. Although the two Raspberry Pi 2 B+ devices were noted to be equally fast for the test processing tasks as the Raspberry Pi 3:s, they were

positioned in the test network topology in such a way that their possible effects on the results would be minimal. In practice this meant using them only when running the tests on more than 8 hubs.

An image serving server was used as the data source for the scripts when they needed to fetch a JPEG image for processing. The image server was implemented as a simple NodeJS application that could serve a whole image, or partial images according to URL parameters. However, when more than 4 hubs were used, one image server could become a bottleneck because multiple hubs were requesting images at the same time. For this reason one image server was set up on each Raspberry Pi to ensure that it would not have an effect on the test results.

The tests were run by using one Raspberry Pi 3 that would run one Solmuhub application (called original hub from now on), one image server, and the test application. The test application is a NodeJS script that sends requests to the original hub and records received responses. It also saves the results from each request to a file for later analysis. The original hub receives the requests, and either processes them locally or distributes them to other hubs when needed. More detailed information about how to setup the testbed and run the tests on it can be found in Appendix 1.

4.3.2 Scenario 1: One-hop parallel distributed computation

One-hop distribution is a simple model of distribution where the processing hubs are directly connected to the original hub. When data needs to be distributed, the original hub maps the data and sends it to the processing hubs for execution, which return their responses back to the original hub. The original hub waits for all responses, and then combines them to produce a complete result, which is finally returned to the test application.

4.3.2.1 Latency

Latency for one-hop URL and data mapped distribution is shown in Figure 10. Interestingly, data mapped execution is 34% faster when using 2 remote hubs, and 20% faster when using 4 hubs, but starts to perform worse when more hubs are added, and is already 8% slower than URL mapped distribution when eight hubs are used. The reason for this is related to the way the data has to be handled in the hubs. When data mapping is used, the initial hub fetches the JPEG image in binary format and converts it to a JavaScript array. This array of pixel data, along

with other metadata related to the request, is then sent to the processing hubs as a JSON string. Because the processing hubs also need the metadata to be able to process the data, the data alone cannot be sent in binary format.

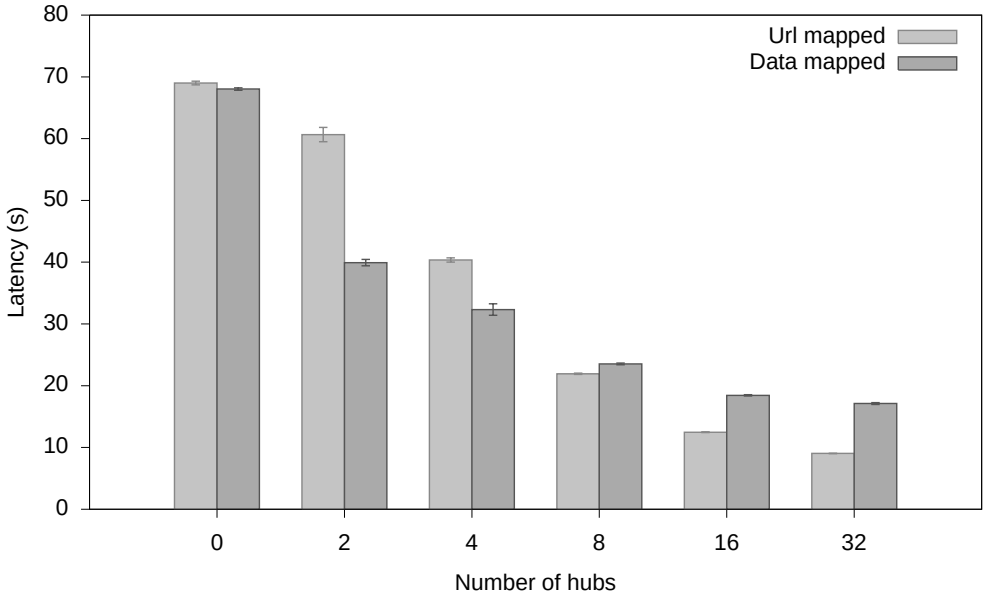


Figure 10: **Mean latency for one-hop execution.** *Latency decreases clearly for both mapping types as more hubs are used. Data mapped distribution is better with small amount of hubs, but URL mapped execution scales better and is faster when more than 4 hubs are used.*

In NodeJS, converting a Buffer object that represents binary data into an array can take multiple seconds, especially in resource constrained devices like the Raspberry Pi. In data mapping this conversion happens only once in the original hub, and the other processing hubs do not have to do it again. However, when URL mapping is used, pieces of data are fetched in the remote hubs, and each hub has to perform the conversion of binary data to a JSON string. When a small number of hubs are used, the binary objects are fairly large, and conversion takes a relatively long time. When more processing hubs are used, converting the data gets faster in each remote hub, and performance starts to surpass data mapped execution, which always has to convert the whole image data in the original hub before offloading.

If binary data was not used, the URL mapped execution should perform better than these tests show. Even though in data mapped execution the initial hub would not have to convert binary data to a JSON string, it would still have to do the data mapping, which is a relatively heavy process compared to URL mapping.

The tests show that when the heavy part of processing can be distributed to remote hubs, processing scales better because it can offload a larger part of the computations to other devices. The theory of Amdahl's law supports this finding, and this also provides a partial answer to research question Q1: if a task has a heavy processing part that can be distributed, then the distributed processing scales well and the part is a good candidate for offloading.

These results also provide an answer to research question Q2: if only a limited amount of hubs are available for distribution, data mapped distribution can be faster than URL mapped, especially when binary data is used. Also, if the remote hubs do not have the data locally accessible, and the data has to be fetched from a central source, it could become a bottleneck in URL mapped distribution.

4.3.2.2 CPU usage

CPU usage is an important factor in the test results, because in addition to directly showing how much CPU time is saved with remote execution, it also gives indirect information about how much energy can be saved, and how different mapping methods affect total energy usage. Because direct energy metering was not in the scope of these tests, CPU usage results were the only way to estimate energy utilization of the IoT Hub.

Figure 11 shows CPU usage in the initial hub during distributed execution, using both URL and data mapping. When processing is not offloaded, CPU usage is about 60%. When distributing the processing to 2 hubs, CPU usage drops to 12% for data mapped execution, and to 0.3% for URL mapped execution. When more hubs are used, both mapping types show increase in the amount of CPU utilized, but data mapping uses clearly more CPU in the original hub than URL mapping. This is because data mapping always uses the same time in the original hub to fetch and map the data, regardless of the amount of remote hubs used, and it cannot be reduced.

Regarding energy usage, it is interesting to note that the amount of hubs used for offloading influences the total energy usage in the original hub only indirectly. Even if only one hub would be used for offloading, the original hub would still benefit from not having to process the data by itself. Thus, the latency of the remote hubs' processing time is the most important factor that affects the total energy usage in the original hub. However, because CPU usage is very low at the original hub during

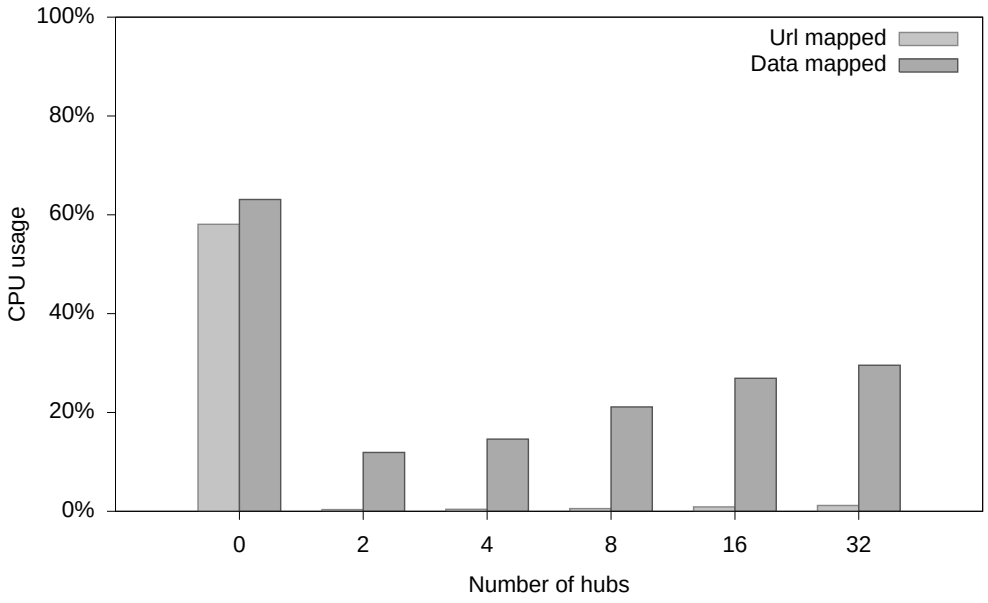


Figure 11: **Mean CPU usage for one-hop execution.** *Mean CPU usage in the original hub drops considerably when processing is offloaded to remote hubs. URL mapped distribution is more economical in CPU usage than data mapped distribution.*

the time when it's waiting for responses from the remote hubs, even a long latency affects the energy usage in the original hub very little.

These results provide answers to research questions Q1 and Q2. For Q1: if energy usage is an important factor, distribution can decrease CPU usage considerably, and thus energy consumption should be lower. Furthermore, even if only a small amount of hubs are available for offloading, it is still beneficial to distribute the processing. For Q2: URL mapped distribution is more effective than data mapped if CPU and energy usage are important, although data mapped distribution can also bring significant savings.

4.3.2.3 Memory usage

Figure 12 shows memory usage for the URL and data mapped execution. Memory usage information is important for IoT devices, because they often cannot use large amounts of memory. It is good to remember that these results only show memory usage using Solmuhub, the NodeJS implementation that uses the V8 JavaScript engine. As the execution engine tests showed, other JavaScript engines such as the Duktape might be more efficient in memory usage, if latency is not a critical issue.

Memory usage for Solmuhubs starts from 17% for URL mapped and 20% for data mapped execution when not offloading the processing, and increases steadily as more remote hubs are utilized. For 32 remote hubs, both URL mapped and data mapped execution use 29% of memory. The increased memory usage for larger number of hubs can be explained by the fact that more data structures have to be managed in memory for handling all the responses from the remote hubs. Also, data mapped execution uses a little more memory than URL mapped, because it needs to handle additional data structures to map the actual data before offloading.

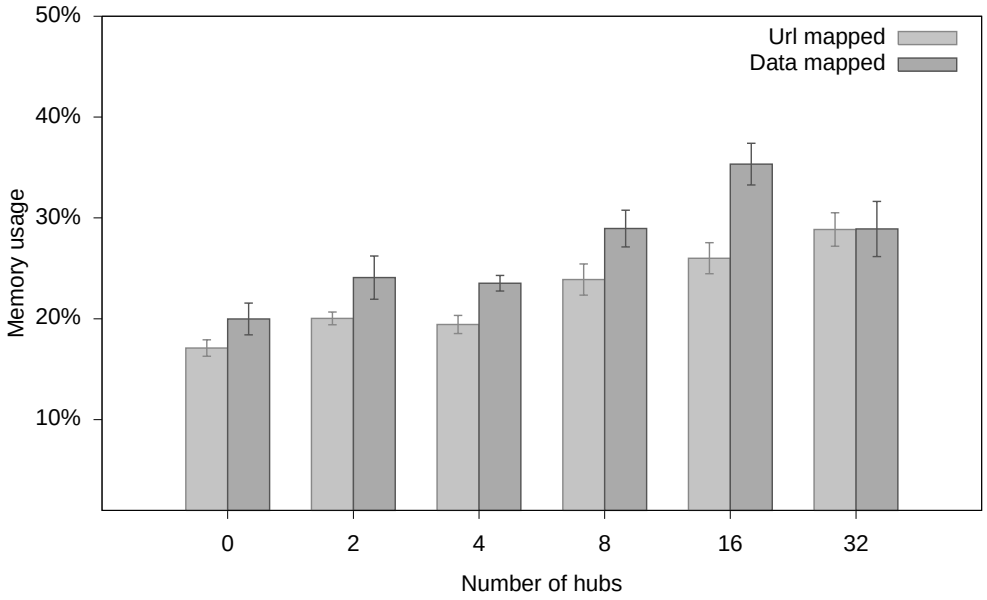


Figure 12: **Mean memory consumption for one-hop execution.** *Memory consumption increases gradually when more hubs are used. When data is processed locally, less memory is used in the hub because data does not need to be mapped and reduced, so there are less large data structures used.*

4.3.2.4 Payload size

Figure 13 shows request payload sizes for both URL and data mapped execution from the original hub to the remote hubs. It shows that URL mapped execution is much more efficient in bandwidth usage for these initial requests. However, when URL mapping is used, the remote hubs still may have to fetch the data by themselves, resulting in a similar amount of total bandwidth usage in the network than when using data mapping. If the remote hubs have access to the data locally, then the total bandwidth usage for requests is as the Figure 13 shows.

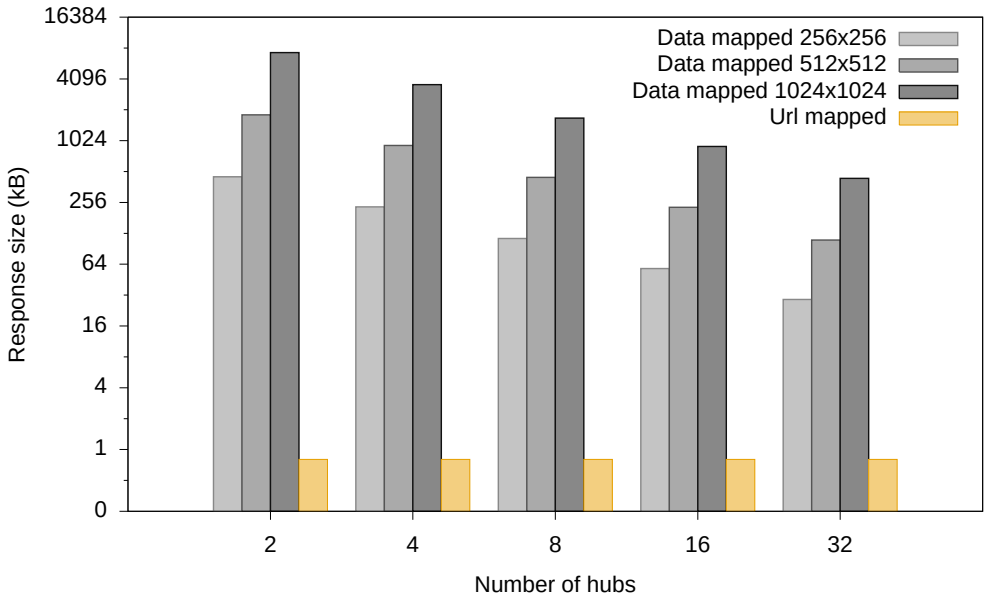


Figure 13: **Request payload sizes in URL and data mapped execution.** *URL mapped offloading uses only a fraction of the bandwidth compared to data mapped offloading. However, if data is not available locally to remote hubs, URL mapping results in a similar amount of bandwidth usage than data mapping.*

Data mapped execution can be seen as using the bandwidth in a more narrow way, sending large amounts of data in a relatively few requests, while URL mapped execution spreads the bandwidth usage larger to the network. If data is available locally in the remote hubs, then URL mapping is clearly more desirable as no data needs to be sent over the network at all.

This request payload size comparison is relevant for congested networks, or networks that do not otherwise have resources to carry large amounts of data. These results give an answer to research question Q2: if network bandwidth usage is to be kept at minimum between the original hub and the remote hubs, URL mapping is a more efficient solution. Also, local data sources should be used when possible, because they result in a greatly reduced network bandwidth usage.

Naturally, also the response sizes of the remote processing affect the total bandwidth usage in the network. However, because the response sizes are identical for URL and data mapped execution, and their sizes cannot be affected, they are omitted in these results. Because response sizes can vary when using multi-hop distribution, they are discussed in Section 4.3.3.

4.3.2.5 Amdahl's law

The Amdahl's law results show how the Solmuhub implementation scales when compared to the theoretical limits of parallel processing. In Figure 14, URL and data mapped execution's performance is presented and can be compared to 80%, 90% and 95% parallelizable processing. The 95% line also presents Solmuhub's theoretical maximum, which was calculated using the profiling information seen in Figure 15. It means that in theory, Solmuhub could achieve a performance gain by the factor of 20, if large enough amount of hubs were used.

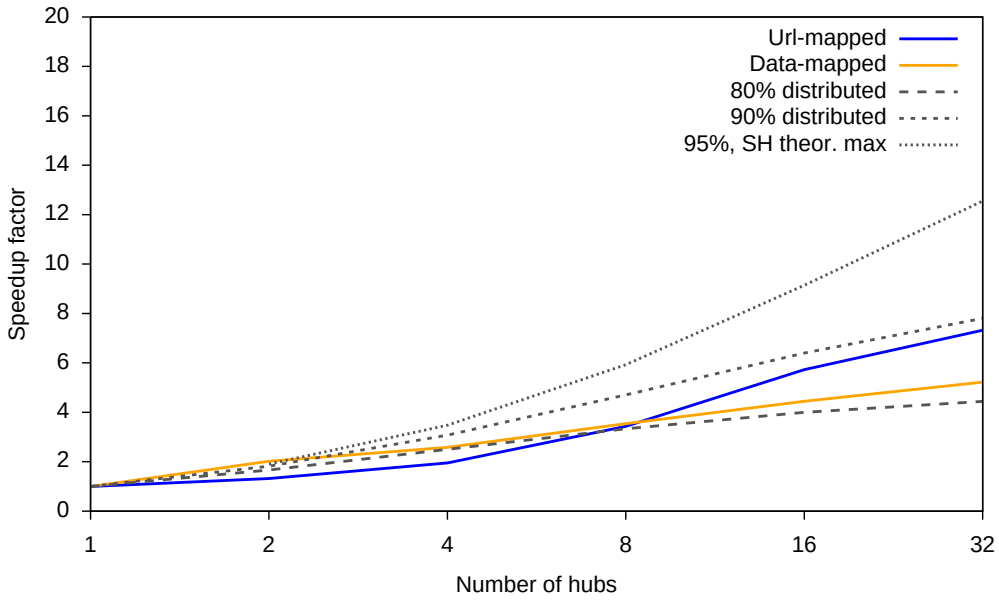


Figure 14: **One-hop execution performance compared to Amdahl's law.** *URL mapped execution's performance increases when more hubs are added. With more than 16 hubs, it is close to 90% parallelizable according to Amdahl's law. Data mapped execution scales worse than URL mapped, staying close to the 80% line.*

In practice, URL mapped and data mapped execution do not reach the 95% parallelizable limit, but rather perform between the 80% and 90% parallelizable limits. URL mapped execution is interesting, because its performance starts even below the 80% line, but when more hubs are added, it starts to improve fast and almost reaches the 90% line when at least 16 hubs are used. Data mapped execution, on the other hand, starts and stays a little over the 80% theoretical line, being at first faster than URL mapped but performing worse when at least 8 hubs are used.

These results mirror the latency results, which already showed that URL mapped execution is slower with smaller amount of hubs, due to the slow binary data con-

version that has to be performed in each remote hub. If no binary conversion would need to be done, URL mapped execution would be at least as fast as data mapped also for the smaller amount of hubs.

Amdahl's law is a good benchmark that simply shows how a distributed system scales. System's performance for different amount of processors can be inspected and anomalies found, like the Solmuhub's performance drop when few hubs are used in URL mapping. Also, when a trend can be seen in the results, a system's performance can be estimated for a larger number of hubs. This is useful in real implementations, because a system can be tested with limited amount of processors, and the decision whether to purchase more processing power can be driven by these results.

4.3.2.6 Profiling data

Profiling information, meaning collecting timestamps in certain parts of the program execution, was collected in each hub so that the individual parts of the program's execution could be timed and their performance evaluated. This was useful for debugging, when anomalies in performance arose and when it was not at first clear why they had occurred. Two examples of such anomalies were the image servers becoming a bottleneck, and the data mapped execution's better performance with a small number of remote hubs. Profiling data was also important when the theoretical limit for Solmuhub's scalability according to Amdahl's law was calculated. It enabled to precisely determine the points in the execution when data was offloaded to remote hubs, so that the local and remote processing times could be calculated.

Figures 15 and 16 show profiling information of URL and data mapped execution, respectively. They present the latency of each program execution part, and show clear differences in the profiles between the two mapping methods. URL mapped execution's profile stays mostly the same when the amount of hubs increases, but data mapped execution's profile clearly changes. With increasing number of remote hubs, data mapped execution's local portion starts to rise. This happens because the total amount of processing time decreases with more hubs, but the time to initially fetch and map the data stays the same. Thus, the remote execution takes a relatively smaller amount of time compared to the total execution time. With URL mapping, the data does not need to be fetched in the original hub, so remote execution continues to take almost all of the total execution time.

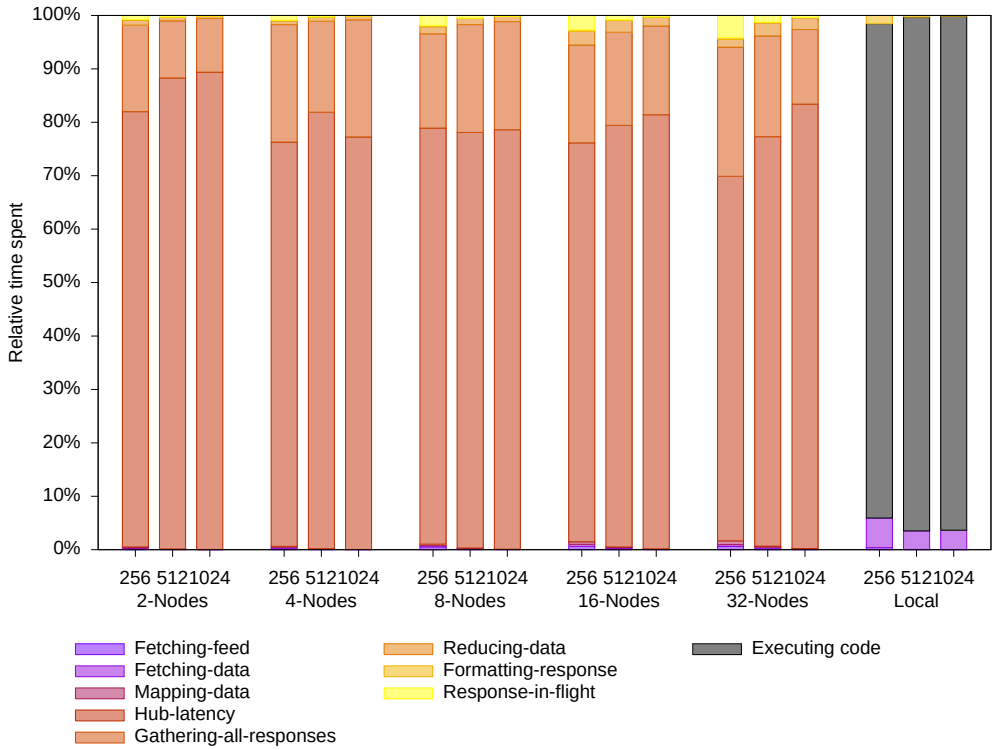


Figure 15: **Solmuhub URL mapped execution profile.** *Remote execution takes over 95% of the time for every type of distribution, so URL mapped execution should scale well for even larger amount of hubs.*

The most significant part that changes in data mapped execution's profile is a block of time that describes the initial fetching of data and mapping of it in the original hub, before sending to remote hubs. The time blocks are increasing because as the total execution latency decreases, a relatively larger amount of time is consumed by the local fetching and mapping time, which stays constant.

URL mapped execution does not have any time consuming local tasks, and so the remote execution takes over 95% of the total execution time. An important thing to notice is that these remote execution percentages cannot be used for calculating the theoretical limits for IoT Hub's performance, because they only show how much time is relatively used in remote processing, and do not take into account absolute latency. So if remote execution takes a long time, the profile for remote processing would show that almost 100% of the time is consumed remotely, leading to a false conclusion that the theoretical limit would be the same. In fact, local execution's

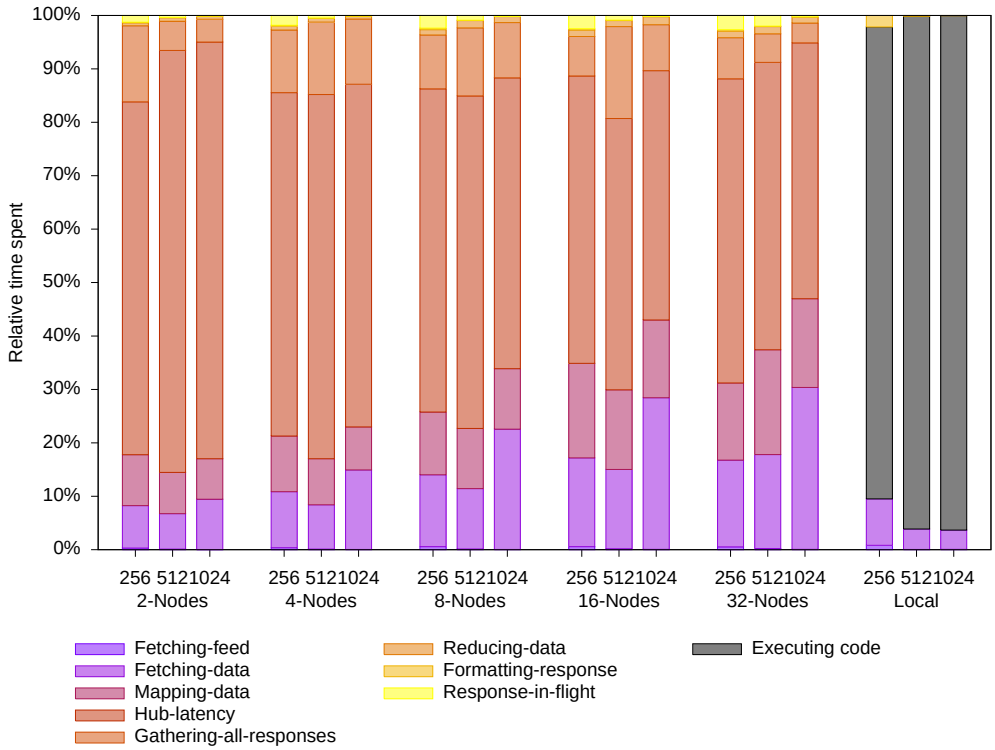


Figure 16: **Solmuhub data mapped execution profile.** *Remote execution takes most part of the total processing time, but the fetching and mapping parts also take a considerable amount of time. Local processing time relative to remote processing time increases when more hubs are added, because total processing time decreases.*

profile has to be used to correctly determine the theoretical limit.

Local execution's profile is also shown in both figures, and it is naturally almost identical in both of them because no distribution occurs. As mentioned, local profile is important because it was used to determine the theoretical limit for IoT Hub's parallel processing using the Amdahl's law. The `Executing code` block in both figures 15 and 16 that shows how large portion was relatively used to execute the script code is the part that in theory could be offloaded to other hubs. Thus, about 95% could be parallelized for IoT Hub execution.

4.3.3 Scenario 2: Hierarchical distributed computation

Hierarchical distributed computation, or multi-hop distribution, means that processing can be further distributed in hubs that already process only partial data.

This naturally adds an overhead to the execution, but there are some scenarios which could still benefit from multi-hop distribution. For example, if only a limited amount of hubs can be made available for initial distribution, but these first level hubs have access to more hubs themselves, it is possible to distribute the processing to many more hubs using multi-hop distribution.

Two different multi-hop scenarios were used: two-hop and three-hop distribution. Both of these distribution types are shown in Figure 9. Regarding the mapping schemes, only URL mapping was used, because of limited time to implement nested execution also for the data mapped distribution.

In the following test results, focus is on comparing the multi-hop distribution with the one-hop distribution, and also the two-hop and three-hop distributions with each other. Some details about the metrics and why they are relevant are not presented here again, as they were already detailed in Section 4.3.2.

4.3.3.1 Latency

Latency for multi-hop distribution is shown in Figure 17. As expected, two-hop execution is slightly faster than three-hop execution. When 8 processing hubs are used, two-hop execution is 15% faster than three-hop, and 17% faster when 16 hubs are used. For 32 processing hubs however, an interesting anomaly is present. The mean latency is 30% higher for two-hop execution than when using 16 hubs, and there is a much larger variance in the results. This is probably because only 32 hubs in total could be used for processing, so some hubs needed to be re-used as intermediate hubs that both took part in the re-distribution of processing, and in the actual payload code execution. This could cause some intermediate hubs to have to wait for some other hubs to finish their processing before they could proceed, incurring delays for the total processing time. The topology of the distribution was tried to be implemented in a way that this overlapping would be minimal, but apparently it could not be avoided in this case. The test was run multiple times, but the results were similar on all occasions. To remove this type of anomaly would require each hub in the distribution process to be used only once in the distribution topology, requiring more actual test devices.

When comparing multi-hop results to one-hop execution, two-hop execution is about 20% slower and three-hop about 25-30% slower. If data mapping was used, the overhead would be higher, because the mapping part would take longer in the in-

termediate hubs, and also more data would have to be sent to the executing hubs. URL mapping is thus more desirable method of distribution in multi-hop scenarios.

These results provide an answer to research question Q3: if many hubs are available for distribution, one-hop distribution is more desirable than multi-hop distribution. However, if only limited amount of hubs are available initially, but those first-level hubs may distribute processing further, then multi-hop distribution can be nearly as effective as one-hop execution.

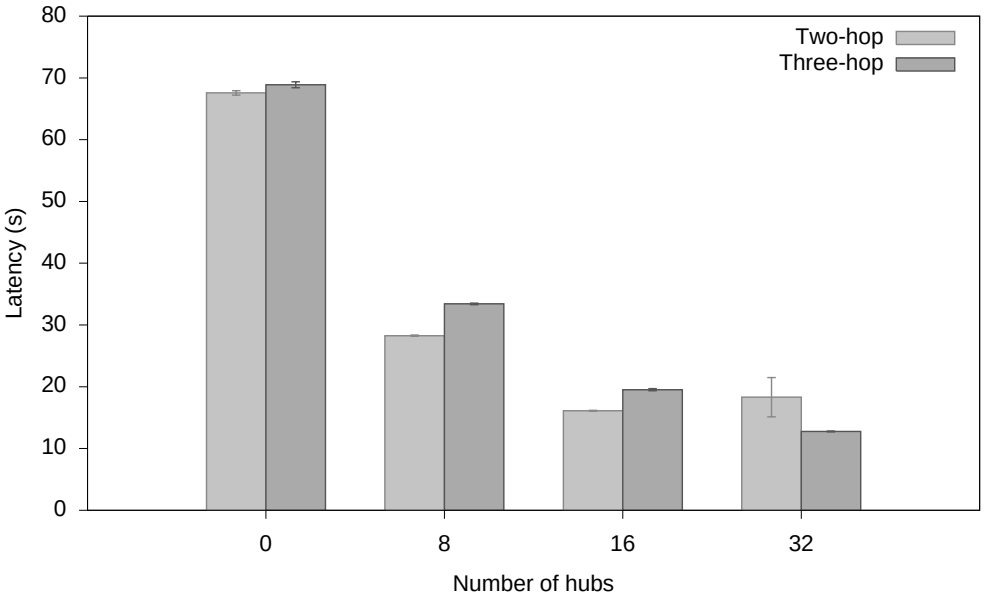


Figure 17: **Mean latency for multi-hop execution.** *Mean latency decreases as more hubs are used. Level 3 distribution is slightly slower than level 2 distribution, but scales well. Overlapping execution hubs cause bigger variance in latency when 32 hubs are used in level 2 distribution.*

4.3.3.2 CPU usage

CPU usage for multi-hop execution is shown in Figure 18. The results are very similar to one-hop execution's results, with CPU usage staying between 0.6% and 0.8% for the two-hop , and between 0.6% and 1.4% for the three-hop execution. These results show that multi-hop execution can save energy in the same manner as one-hop execution, because the original hub is not affected by how deeply the computation is distributed. Similarly to one-hop execution, energy used in the original hub is always saved by offloading computations.

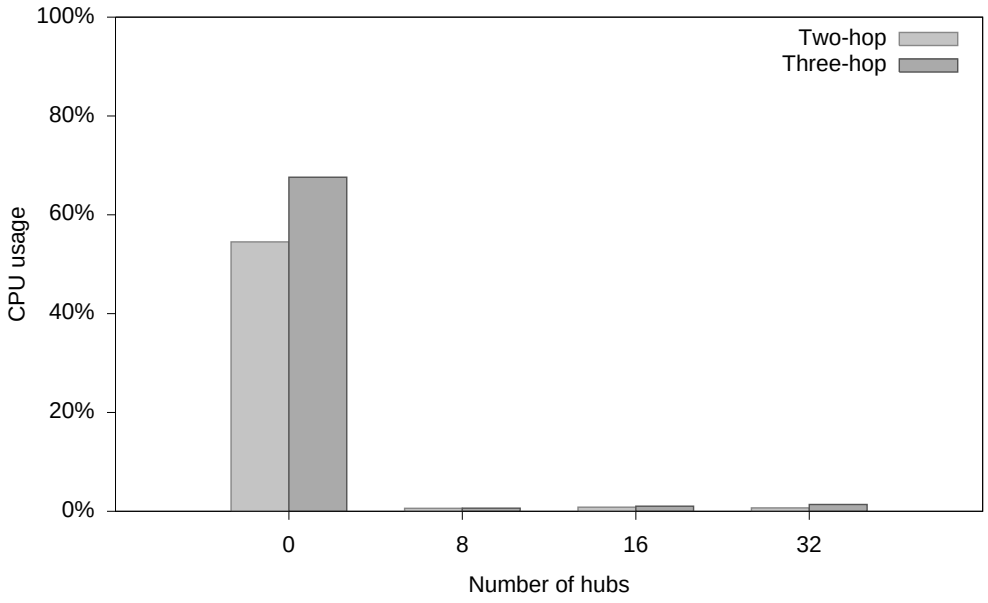


Figure 18: **Total mean CPU usage for multi-hop execution.** *Mean CPU usage drops dramatically when distributing execution. CPU usage is lower than in one-hop distribution, because even less requests have to be sent initially.*

For the distribution part, mapping occurs in the same way as for the one-hop execution, and because there are less hubs used for the first level of distribution, even less requests have to be sent to the hubs in the next level of distribution. This can reduce the local processing time a little, but remote processing latency affects mean total CPU usage much more. When remote processing takes a long time, it decreases the mean CPU usage because local CPU usage in the original hub is very low during that time.

4.3.3.3 Memory usage

Figure 19 shows the memory consumption for multi-hop execution. When comparing multi-hop to the one-hop execution, memory usage is higher for the multi-hop. One-hop execution's memory usage from Figure 12 shows that memory consumption for URL mapped method is between 23% and 29%. While two-hop execution uses 23% for 8 processing hubs, the consumption rises to 35% for 16 and 39% for 32 processing hubs. Three-hop execution's memory usage stays more constant than two-hop's, using 28%, 31% and 30% for 8, 16 and 32 processing hubs, respectively. So between the two models, memory consumption varies: two-hop execution uses 6 percentage points (pp) less memory for 8 processing hubs, but 4.5 pp and 9.5 pp

more than three-hop when using 16 and 32 processing hubs.

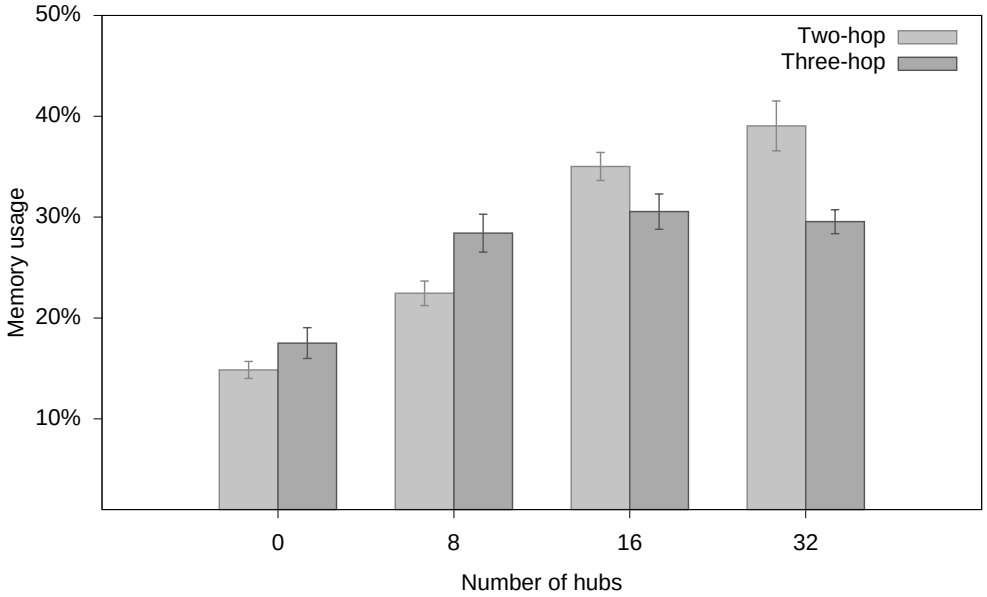


Figure 19: **Mean memory consumption for multi-hop execution.** *Memory consumption increases when more hubs are used. Level 3 execution uses less memory for larger amount of hubs, because it uses less hubs in level 1.*

As noted, memory usage for multi-hop execution increases along with the number of remote hubs used. The reason for this is probably because the number of response messages also increases when more hubs are used. The number of response messages rises because the more hubs are used for processing in third-level, the more hubs are also used for initial first-level distribution. The number of response messages to the original hub thus directly correlates with the number of first-level hubs. Also, because there are more response messages for two-hop execution than for three-hop, it explains why the two-hop execution's memory usage increases more than the three-hop one's.

As a last note on memory usage, the implementation of Solmuhub is currently not optimized for memory consumption, and analyzing and improving it further could bring memory usage down considerably. Also, these results show that the topology of the network used for distributing the processing might affect memory usage. However, this hypothesis would require more inspection and further tests to verify. The different sizes of the response messages in multi-hop execution are discussed next, and should further clarify the issue.

4.3.3.4 Payload size

Payload sizes of the response messages in multi-hop execution is depicted in Figure 20. The figure shows how response sizes between the two multi-hop models vary. The response sizes for two-hop execution are 8MB, 4MB and 2MB respectively for 8, 16 and 32 leaf-level hubs. On the other hand, three-hop execution's responses are sized 16MB, 8MB and 4MB for similar amount of leaf-level hubs. So it seems that the total response sizes for two-hop and three-hop executions are different for the same amount of hubs. However, the number of hubs presented in the Figure 20's X-axis shows how many hubs are actually processing at the leaf level, which is not the same as the total number of hubs, or the number of hubs in the first level of distribution. For example, a two-hop execution having eight hubs in the leaf level only has two hubs at the first level of distribution. Similarly, for three-hop execution, eight hubs in the leaf level results in using only one hub in the first level, four hubs in the second level, and eight hubs in the last third level of distribution.

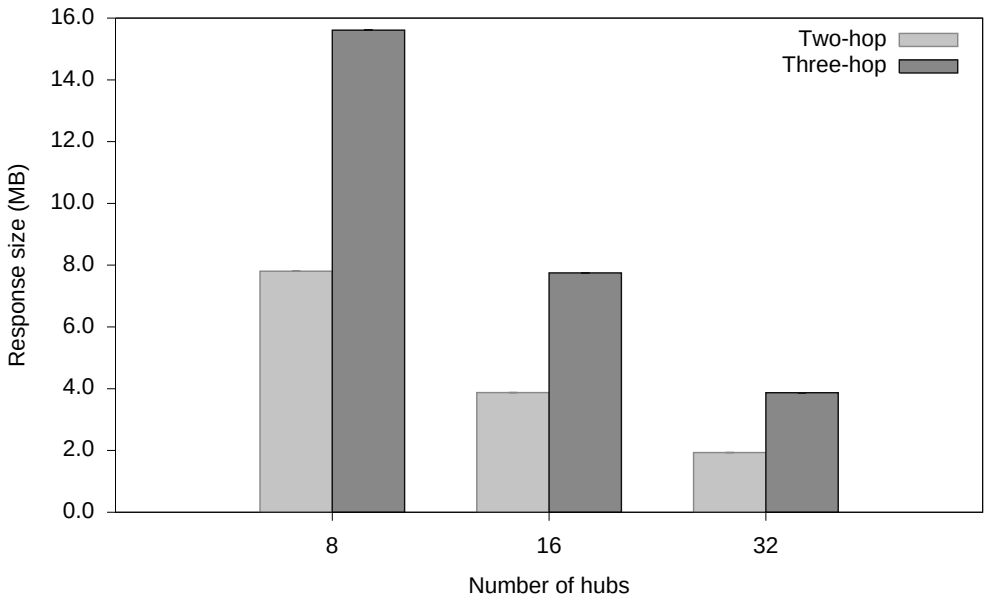


Figure 20: **Mean payload size of solmuhub response messages for two-hop execution.** *Response payloads in multi-hop execution are larger compared to similar amount of processing hubs for one-hop, because there are less hubs returning results from the first level of distribution.*

The most important thing to understand about these payload results is that although an equal amount of hubs are used in the leaf level, the sizes of the response messages can vary. They depend on the topology of the distribution: how many hubs are

used in each level of distribution. The fewer hubs there are used at the first level of distribution, the larger the individual responses are. Importantly, as noted before, the topology of the distribution may effect memory consumption in the original hub, which may be of concern when considering a topology for bandwidth constrained networks.

4.3.3.5 Amdahl's law performance

Amdahl's law results are shown in Figure 21. Both multi-hop scenarios are quite close to each other in terms of performance. Two-hop distribution is constantly a little better than three-hop, as expected, but not by much. The speedup factors for two-hop execution are 2.7 for 8 hubs, 4.6 for 16 hubs and 6.3 for 32 hubs. The corresponding values for three-hop execution and similar amount of hubs are 2.3, 3.9 and 5.6.

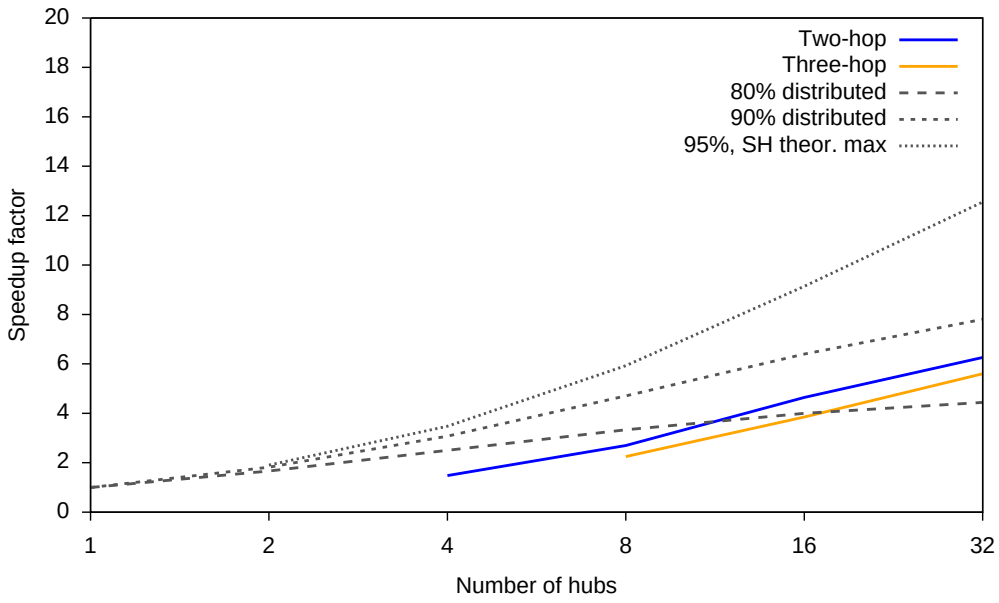


Figure 21: **Multi-hop execution performance compared to Amdahl's law.** *Performance scales well, and is between the URL mapped and data mapped execution's performance when using at least 16 hubs.*

Comparing to theoretical values, both types of execution stay below the 80% parallelizable line according to Amdahl's law, but scale surprisingly well. When 32 hubs are used at the leaf level, they perform even better than one-hop data mapped execution for the same amount of hubs, which is remarkable. The performance trend

also seems to steadily increase, so testing with larger amount of hubs could show further benefits.

These results are important because they show that multi-hop execution does naturally include an overhead compared to the one-hop execution, but that it still scales well and can even be faster than the data mapped one-hop execution, if sufficiently large amount of hubs are used.

4.3.3.6 Profiling results

Profiling information for multi-hop execution is shown in figures 22 and 23. The profiles are similar to URL mapped one-hop execution's respective profile, but some differences can be found, especially when using larger amount of hubs. When at most 16 hubs are used, multi-hop execution uses clearly less time for gathering responses, which is because less responses are returned from the first level of hubs than in one-hop execution for similar amount of processing hubs.

Another notable difference is that when using two-hop execution and 32 hubs for processing, the gathering-responses block is larger than in three-hop or one-hop execution. This actually shows the variance that occurred in two-hop execution due to overlapping with intermediate, re-distributing hubs and code executing leaf hubs.

The profiling results show that multi-hop execution uses relatively less time for mapping and reducing data to and from the remote hubs than one-hop execution. This is good to remember when considering different topologies for distribution. If local execution is to be kept at minimum, multi-hop execution can offer fast mapping of data to the few initial, first level hubs, which on their part can further distribute the processing to another level of hubs. However, multi-hop execution incurs a little latency overhead, which might or might not be relevant, depending on the scenario.

4.4 Summary

Distributed processing tests using Solmuhub show that IoT Hub implementation can offload computations effectively. When sufficient amount of remote hubs are used, great benefits can be achieved in terms of latency, CPU/energy usage and network bandwidth. Memory usage does not seem to benefit as clearly as the other metrics, but this might be possible to change with better memory management in future versions of the application.

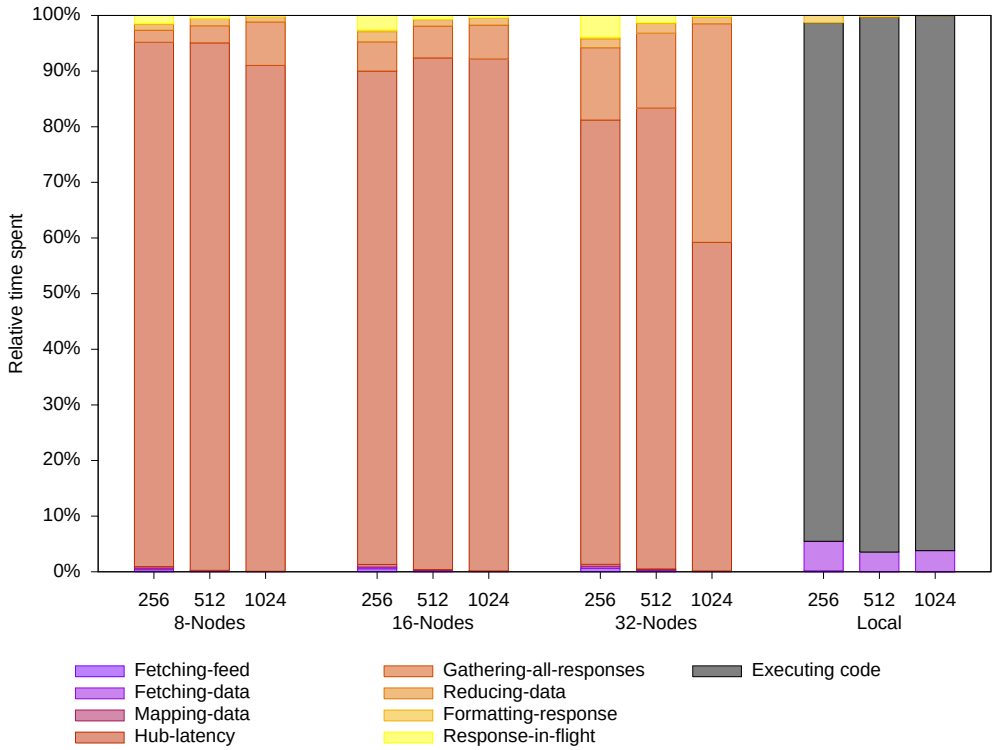


Figure 22: **Solmuhub processing profile for two-hop execution.** *Gathering responses takes a relatively smaller amount of time compared to one-hop execution, because there are less hubs in the first level of distribution for two-hop execution.*

URL mapped versus data mapped distribution gave evidence that while data mapping might be worth considering for smaller amount of hubs, URL mapped execution scales better. Specifically, if large amount of hubs are available for distribution and the data that is used in processing is locally available in each remote hub, URL mapping is superior to data mapping. However, if the payload data requires preprocessing before it is sent to the network, then data mapping can be faster when smaller amount of hubs are used.

Nested execution tests showed that multi-hop distribution can be a competitive way to offload processing in some scenarios. Specifically, if a relatively few number of hubs are available for the first level distribution, but those hubs can distribute the processing further, then multi-hop distribution can be a good choice. The tests showed that a latency overhead of 20-30% was added when using two- and three-hop execution.

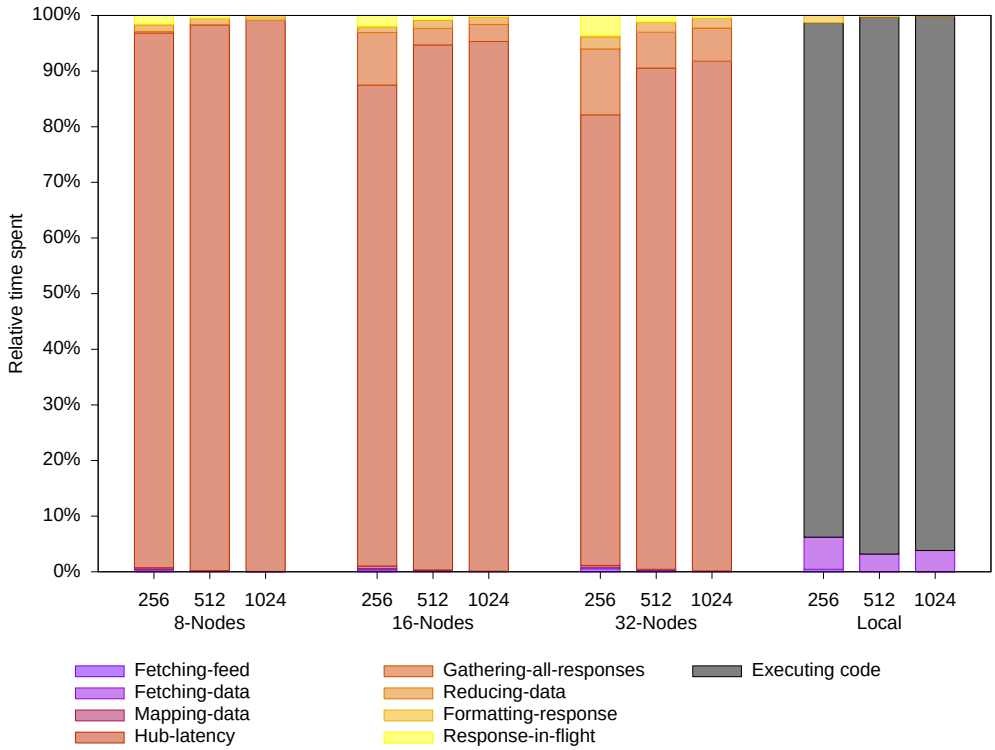


Figure 23: **Solmuhub processing profile for three-hop execution.** *Results are similar to two-hop execution. Less time is relatively used to wait for all responses in the original hub, because there are fewer requests sent to the first level of hubs than when using one-hop or two-hop execution.*

5 Conclusions

This thesis has discussed distributed computing on edge devices, which will be an important form of computing in the near future due to the massive increase in the amount of data in the edge networks. The traditional way to cope with high processing requirements e.g. in mobile phones has been to utilize cloud computing for offloading tasks. However, utilizing remote clouds inherently introduces high latency, which is intolerable for many IoT applications. More recent approaches include the fog computing architecture, which targets to offload tasks to local computing resources. This is a better approach, but requires purposeful machines in the environment to be setup. This thesis argues that the IoT devices themselves can be utilized to process offloaded tasks, thus harvesting the existing infrastructure of

computing devices in the edge networks.

Current IoT systems offer APIs for monitoring, management and control of the devices, but they do not have a standardized way to expose the compute resources of these devices. The proposed solution, the IoT Hub platform, presents a software layer that can be used to access the processing power of the IoT resources. The platform can be deployed on existing IoT devices and middleware with the help of enablers, and provides a generic RESTful API that handles all communication between different IoT Hubs. Also, distributed offloading is possible so that the tasks are small enough to be executed on resource constrained devices.

By running tests of distributed tasks on an actual IoT Hub testbed, it has been shown that the IoT Hub can be used to distribute computing tasks to a large number of IoT devices, and is capable of executing them in parallel. The test results also show that distributed processing on the platform can reduce CPU utilization and latency considerably compared to executing on a single device. Thus, it can be used to both save energy and get the results faster. Also, larger data sets become viable for processing on constrained devices, because only a fraction of the data has to be processed on any one hub.

IoT Hub performance for one-hop execution was evaluated using two mapping schemes: URL mapping and data mapping. For the most part, URL mapping was found to be more performant, although in situations when only a few hubs were used for distribution, data mapping was faster. URL mapping also scaled better, and using more hubs increased the system's relative performance when Amdahl's law was used as a benchmark. Data mapped execution scaled more linearly, and adding more hubs did not incur as big a performance boost as it did for URL mapping.

One-hop execution using URL mapping was compared to two- and three-hop execution, and it was shown that multi-hop execution can also reduce latency significantly, while enabling wider distribution in situations where only a limited amount of hubs are initially available to the original hub for offloading. However, multi-hop execution always introduces an overhead, which is between 20-30% for two- and three-hop execution.

In the future, further testing could be implemented to include multi-hop scenarios also for data mapped distribution, which could then be compared to the corresponding URL mapped multi-hop execution. To get more accurate results for energy consumption, energy usage measuring of the original hub would have to be implemented. For the execution engines, it would be interesting to compare multi-

ple different execution engines for the payload code execution in the NodeJS version of the IoT Hub. In addition to enabling more granular memory usage analysis, it would allow a more precise comparison of the Java implementation with the NodeJS one, as both could use the Duktape engine for the payload code execution.

While these tests did not include devices with higher processing power, it would be interesting to compare a basic laptop's or server's performance to a group of IoT devices. Yet another future possibility would be to compare distributing processing to local resources versus distributing to the cloud, as that would give a concrete proof of how much time can be saved by utilizing edge resources.

References

AA16 Ahmed, A. and Ahmed, E., A survey on mobile edge computing. *2016 10th International Conference on Intelligent Systems and Control (ISCO)*. IEEE, 2016, pages 1–8.

AFG+10 Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. et al., A view of cloud computing. *Communications of the ACM*, 53,4(2010), pages 50–58.

ÅGB11 Åkerberg, J., Gidlund, M. and Björkman, M., Future research challenges in wireless sensor and actuator networks targeting industrial automation. *2011 9th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 2011, pages 410–415.

AIM10 Atzori, L., Iera, A. and Morabito, G., The Internet of Things: A survey. *Computer networks*, 54,15(2010), pages 2787–2805.

Amd67 Amdahl, G. M., Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), New York, NY, USA, 1967, ACM, pages 483–485, URL `http://doi.acm.org/10.1145/1465482.1465560`.

Bak00 Baker, M., Cluster computing white paper. *arXiv preprint cs/0004014*.

BFG+00 Brightwell, R., Fisk, L. A., Greenberg, D. S., Hudson, T., Levenhagen, M., Maccabe, A. B. and Riesen, R., Massively parallel computing using

commodity components. *Parallel Computing*, 26,2-3(2000), pages 243 – 266. URL http://www.sciencedirect.com/science/article/pii/S0167819199001040.

BMZA12 Bonomi, F., Milito, R., Zhu, J. and Addepalli, S., Fog computing and its role in the Internet of Things. *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, New York, NY, USA, 2012, ACM, pages 13–16, URL http://doi.acm.org/10.1145/2342509.2342513.

C+05 Council, N. R. et al., *Getting up to speed: The future of supercomputing.* National Academies Press, 2005.

CDK05 Coulouris, G. F., Dollimore, J. and Kindberg, T., *Distributed Systems: concepts and design.* Pearson Education, 2005.

CZ16 Chiang, M. and Zhang, T., Fog and IoT: An overview of research opportunities. *IEEE Internet of Things Journal*, 3,6(2016), pages 854–864.

DG04 Dean, J. and Ghemawat, S., MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004, USENIX Association, pages 10–10, URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

DLNW13 Dinh, H. T., Lee, C., Niyato, D. and Wang, P., A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13,18(2013), pages 1587–1611.

dRSGS16 del Rio, F. D., Salmeron-Garcia, J. and Sevillano, J. L., Extending Amdahl's Law for the cloud computing era. *Computer*, 49,2(2016), pages 14–22.

EM94 Elzen, B. and Mackenzie, D., The social limits of speed: development and use of supercomputers. *IEEE Annals of the History of Computing*, 16,1(1994), pages 46–61.

Eri11 Ericsson, More than 50 billion connected devices - taking connected devices to mass market and profitability. Technical Report, February 2011. URL http://www.ericsson.com/res/docs/whitepapers/wp-50-billions.pdf.

Eva11 Evans, D., The Internet of Things how the next evolution of the Internet is changing everything. Technical Report, Cisco, April 2011. URL http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.

FK03 Foster, I. and Kesselman, C., What is the Grid. *A three point checklist*, 20.

FLR13 Fernando, N., Loke, S. W. and Rahayu, W., Mobile Cloud Computing. *Future Gener. Comput. Syst.*, 29,1(2013), pages 84–106. URL http://dx.doi.org/10.1016/j.future.2012.05.023.

FOLR85 Fox, G., Otto, S., Lyzenga, G. and Rogstad, D., The Caltech concurrent computation program-Project description.

FZRL08 Foster, I., Zhao, Y., Raicu, I. and Lu, S., Cloud computing and grid computing 360-degree compared. *Grid Computing Environments Workshop, 2008. GCE '08*, Nov 2008, pages 1–10.

Gar14 Gartner says a thirty-fold increase in Internet-connected physical devices by 2020 will significantly alter how the supply chain operates. URL http://www.gartner.com/newsroom/id/2688717.

GLME+15 Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P. and Riviere, E., Edge-centric computing: Vision and challenges. *SIGCOMM Comput. Commun. Rev.*, 45,5(2015), pages 37–42. URL http://doi.acm.org/10.1145/2831347.2831354.

Gus88 Gustafson, J. L., Reevaluating Amdahl's Law. *Commun. ACM*, 31,5(1988), pages 532–533. URL http://doi.acm.org/10.1145/42411.42415.

HPR+13 Ha, K., Pillai, P., Richter, W., Abe, Y. and Satyanarayanan, M., Just-in-time provisioning for cyber foraging. *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 2013, pages 153–166.

HV15 Haubenwaller, A. M. and Vandikas, K., Computations on the edge in the Internet of Things. *Procedia Computer Science*, 52, pages 29–34.

JPS13 Jhawar, R., Piuri, V. and Santambrogio, M., Fault tolerance management in cloud computing: A system-level perspective. *IEEE Systems Journal*, 7,2(2013), pages 288–297.

KBU15 Krishnan, Y. N., Bhagwat, C. N. and Utpat, A. P., Fog computing - Network based cloud computing. *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*, Feb 2015, pages 250–251.

Kel16 Kellinsalmi, N., Mobiili pilvilaskenta elakoittaa konesalit. URL `http://www.aka.fi/fi/akatemia/media/Ajankohtaiset-uutiset/2016/mobiilipilvilaskenta-elakoittaa-konesalit2/`.

KLD12 Kovatsch, M., Lanter, M. and Duquennoy, S., Actinium: A RESTful runtime container for scriptable Internet of Things applications. *Proc. IOT 2012*, 2012, pages 135–142.

Kri10 Kristensen, M. D., Scavenger: Transparent development of efficient cyber foraging applications. *2010 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, March 2010, pages 217–226.

Krz16 Krzanich, B., Data is the new oil in the future of automated driving. URL `https://newsroom.intel.com/editorials/krzanich-the-future-of-automated-driving/`.

Lea13 Lea, R., Hypercat: an IoT interoperability specification, 2013.

LGL+15 Luan, T. H., Gao, L., Li, Z., Xiang, Y. and Sun, L., Fog computing: Focusing on mobile users at the edge. *CoRR*, abs/1502.01815. URL `http://arxiv.org/abs/1502.01815`.

Mac91 Mackenzie, D., The influence of the Los Alamos and Livermore National Laboratories on the development of supercomputing. *Annals of the History of Computing*, 13,2(1991), pages 179–201.

Mar09 Marinelli, E. E., Hyrax: cloud computing on mobile devices using MapReduce. Technical Report, DTIC Document, 2009.

Mat09 Mattson, T., How to sound like a parallel programming expert part 2: parallel hardware. Technical Report MSU-CSE-06-2, Intel, August

2009. URL https://software.intel.com/sites/default/files/m/d/4/1/d/8/09MC04_Sound_Like_PP_part2.pdf.

MG11 Mell, P. and Grance, T., The NIST definition of cloud computing.

MMST16 Mineraud, J., Mazhelis, O., Su, X. and Tarkoma, S., A gap analysis of Internet-of-Things platforms. *Computer Communications*, 89, pages 5–16.

MT15 Mineraud, J. and Tarkoma, S., Toward interoperability for the Internet of Things with meta-hubs. *arXiv preprint arXiv:1511.08063*.

Sat01 Satyanarayanan, M., Pervasive computing: Vision and challenges. *IEEE Personal communications*, 8,4(2001), pages 10–17.

SBCD09 Satyanarayanan, M., Bahl, P., Caceres, R. and Davies, N., The case for VM-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8,4(2009).

SK11 Sadashiv, N. and Kumar, S., Cluster, grid and cloud computing: A detailed comparison. *2011 6th International Conference on Computer Science Education (ICCSE)*, Aug 2011, pages 477–482.

STP+12 Shi, J. Y., Taifi, M., Pradeep, A., Khreishah, A. and Antony, V., Program scalability analysis for HPC cloud: Applying Amdahl's Law to NAS benchmarks. *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov 2012, pages 1215–1225.

VRM14 Vaquero, L. M. and Rodero-Merino, L., Finding your way in the fog: Towards a comprehensive definition of fog computing. *SIGCOMM Comput. Commun. Rev.*, 44,5(2014), pages 27–32. URL http://doi.acm.org/10.1145/2677046.2677052.

YHQL15 Yi, S., Hao, Z., Qin, Z. and Li, Q., Fog computing: Platform and applications. *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. IEEE, 2015, pages 73–78.

YLL15 Yi, S., Li, C. and Li, Q., A survey of fog computing: Concepts, applications and issues. *Proceedings of the 2015 Workshop on Mobile Big Data*, Mobidata '15, New York, NY, USA, 2015, ACM, pages 37–42, URL http://doi.acm.org/10.1145/2757384.2757397.

ZCP+13 Zhu, J., Chan, D. S., Prabhu, M. S., Natarajan, P., Hu, H. and Bonomi, F., Improving web sites performance using edge servers in fog computing architecture. *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, March 2013, pages 320–323.

Appendix 1. Testbed Setup

A description of the testbed and how it can be setup to run the Solmuhub tests is presented here.

To setup an identical testbed to the one used in the distributed tests, following items are required: a 24-port Ethernet switch, 17 Raspberry Pi 3 computers and 17 network cables to connect the Raspberry Pis to the switch. Also, it helps to have one external computer connected to the switch, so that configuring and starting of the Solmuhubs and image servers in the Raspberry Pis is possible. Of course, it is also possible to run the tests on a smaller or larger number of hubs. A picture showing the testbed setup is presented in Figure 24.

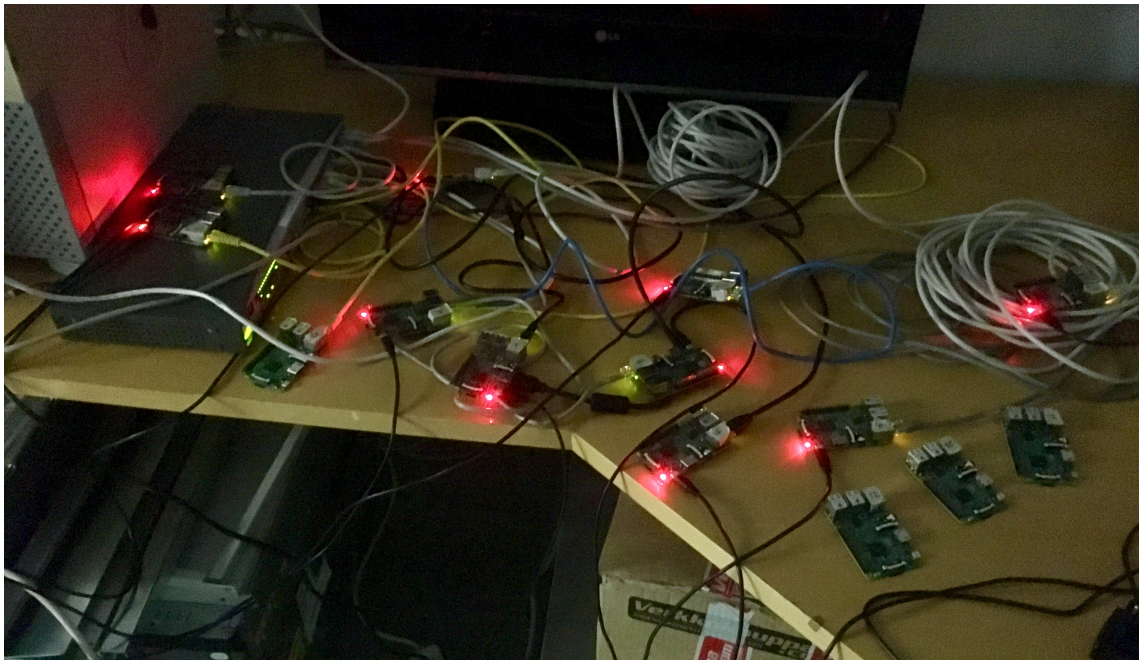


Figure 24: **IoT Hub testbed setup** *The picture shows a 24-port HP switch and multiple Raspberry Pis connected to it with Ethernet cables. The hubs showing a red light are on.*

The individual requests sent by the test code follow the flows shown in Figure 3. For the sake of clarity, the computers in the testbed are given the following roles: the external computer used to SSH to the other devices has the role `helper`, the original device that runs the test script and is sent the first request has the role `controller`, and the devices doing most of the processing have the role `executor`. The amount of hubs used for processing varies from 0 to 32, where 0 means that the

initial processing request is only sent to the `controller`'s IoT Hub, and the payload code is executed locally on that hub. If the amount of hubs used for processing is greater than 0, then the initial request is still sent to the `controller`'s IoT Hub, but it distributes the processing to the `executors`' IoT Hubs.

To run the tests, Solmuhub has to be installed on each IoT Hub (Raspberry Pi) that is used in the testbed. The Solmuhub code can be found from `https://github.com/uh-cs-iotlab/solmuhub`. Also, an image-serving NodeJS application needs to run during the tests on each hub, and the code for that application is available at: `https://github.com/jphire/iothub-data-server`. Finally, the testbed code that is used to run the tests has to be installed on the /textttcontroller hub. The code for the testbed is available at: `https://github.com/jphire/iothub-testbed`. Each aforementioned application includes instructions in their own repository for installing and running them, but the commands needed to run the tests are also presented here. Note that the below instructions can become obsolete as the testbed and Solmuhub applications are further developed, so those applications' online repositories are the best places to seek for more up to date information.

1. Connect the helper and the controller to the switch and turn them on.
2. Setup SSH connection from the helper to the controller, and take note of the controller's IP address and save it, it is needed later.
3. In the controller, install the Solmuhub, the image server and the testbed applications.
4. Start the image-server application by following instructions in its repository.
5. Start the Solmuhub application, and specify a reasonable port (e.g. 3000), and set the http-mode and profiler on. Instructions for starting the Solmuhub with different configurations is provided in its repository.
6. Create an executable feed to the Solmuhub that was started. The name of the feed does not matter.
7. Leave the SSH connection open to the controller.
8. Connect an executor device to the switch, and turn it on. Check the helper for the new devices IP address and save it.

9. SSH to the newly connected executor, and install the image server and the Solmuhub applications to it just like they were installed to the controller.

10. Start the image server in the executor, then start two different instances of the Solmuhub on different ports (e.g. 3000 and 3100) with the profiler on. Create executable feeds to both Solmuhub instances.

11. Now, you should be able to send a request from the controller to the one executor that is set up. It is good to test the connection at this point e.g. by using `curl`.

12. The next step is to repeat the steps done for setting up the one `executor` node to each other `executor` in the testbed. Be sure to add them one by one, so it is easy to take note of each device's IP address.

13. After each `executor` is set up, meaning they all have an image-server and two Solmuhub instances running, the testbed application in the `controller` has to be configured. For this, a list of all the `executors`' IP addresses is needed. When the tests are run, the testbed application first reads the *conf.json* file, which, among other things, specifies the request file that is used as the payload for the initial request to the `controller's` Solmuhub application. Each `executor`'s IP address that is used in the test has to be stored in that request file, or the tests will fail.

14. When the configuration for the testbed is ready, the tests can be run in the `controller` by going to the testbed application's folder and running `node test.js`. The test script will start sending requests, and records the results in the *logs* directory.

After the test are run, the raw results are located in the `logs` directory in the `controller` device. Now, these results can be used to create the figures that were presented in this thesis. The testbed repository includes scripts for generating the figures, and the instructions for their usage.

Appendix 2. IoT Hub Poster

Egde computing with IoT hubs

Janne Laukkanen
University of Helsinki
Julien Mineraud
University of Helsinki
Sasu Tarkoma
University of Helsinki

MOTIVATION

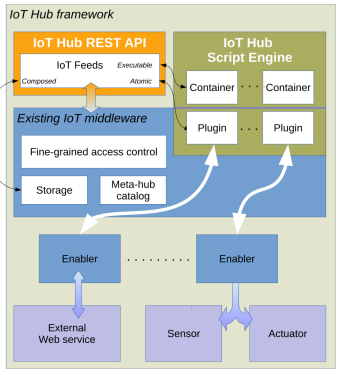
- Digitalization of services requires gigantic volumes of data to be processed in real-time
- The Internet of Things (IoT) accentuates this phenomenon
- This drives network operators to move more intelligence toward the edge of the network
- Our aim is to enable edge IoT platforms to process locally the data they produce

BACKGROUND

IoT platforms are key components to connect to devices, store and retrieve data
- Usually one-hop away (low latency)
- Simple REST APIs to access devices and data
- Access control

However, edge IoT platforms lack fundamentals for local computing
- IoT devices/hubs may have limited computing resources
- Drives the need for distributed computing at the edge

IoT Hub



We developed the IoT Hub platform to fill in the gaps within the current landscape of IoT platforms [1]
- Interoperability
- Fine-gained access control
- Support for IoT developers
- Ecosystem support with meta-hubs [2]

The IoT Hub plugin engine can be used for local computation tasks
- Computation resources are provided via the REST API along with other resource types (data & devices)

LOCAL ENGINES

Goal
- Evaluating performances of selected IoT Hub script engine implementations
- Preferred scripting language is JavaScript (widely used in Web development)

Implementations
- Google's V8 engine (native in Node.js)
- Duktape (optimized for extremely constrained devices)

Results
- V8 and Duktape tested with Newton, Quicksort and Fibonacci problems
- V8 is clearly faster (100x)
- Duktape can run on highly constrained device (platforms with at least 192kB flash and 64kB system RAM)

EXECUTION FEED

Goal
- Exposing computing resources to third parties via the REST API
- New feed type: the execution feed

Implementation
- Extend the IoT Hub script engine to support sand-boxed execution of IoT Hub scripts in containers
- Execution feeds setup is similar to other type of feeds (atomic & composed), and under the control of the hub owner
- Execution feeds' capabilities are defined by the hub owner
  - Max memory
  - Max CPU usage
  - Loaded script libraries

Usage
- Execution feed user sends script to be executed as payload to the feed URI
- Result of the execution feed are send back to the user after completion via the HTTP response

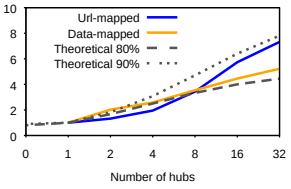
DISTRIBUTED COMPUTATION

Goal
- Leverage meta-hub information to distribute computation task across hubs

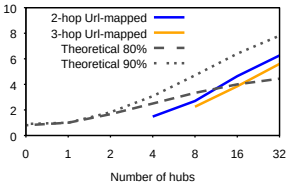
Implementation
- IoT Hub finds external executable feeds from its meta-hub catalog
- Computation is parallelized using the discovered executable feeds
- Data needed for the task can be distributed via payload or an URL for a feed containing the data

EVALUATION

Scenarios
- Url-based (using feeds) vs Data in payload
- One-hop vs. multi-hop (hubs can further distribute their task to other hubs)



(a) one-hop distribution scheme



(b) multi-hop distribution scheme

Figure: Amdahl's law and distribution schemes: (a) one-hop and (b) multi-hop

Amdahl's law describes how parallel computing can theoretically scale with respect to how large are the parts of the program execution that can be parallelized.

Results
- Both distribution schemes exhibit good parallelization performances (between 80% and 90% of the theoretical limits defined by the law)
- One-hop distribution scheme is the most efficient, but requires all hubs to be visible to the hub initiating the computation task
- Multi-hop scheme introduces a latency overhead but is more flexible as more hubs may be available for computation

FUTURE WORK

- Monitoring of IoT hub performances for efficient task distribution
- Spark-like evolution of the IoT hub
- Privacy awareness while sharing data
- Virtual feeds

REFERENCES

[1] J. Mineraud, O. Mazhelis, X. Su and S. Tarkoma, A gap analysis of Internet-of-Things platforms, Computer Communications, Volumes 89–90, 1 September 2016, Pages 5-16
[2] J. Mineraud and S. Tarkoma, Toward interoperability for the Internet of Things with meta-hubs, http://arxiv.org/abs/1511.08063

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI
MATEMAATTIS-LUONNONTIETEELLINEN TIEDEKUNTA
MATEMATISK-NATURVETENSKAPLIGA FAKULTETEN
FACULTY OF SCIENCE